

MARCO CANTÙ

OBJECT PASCAL HANDBOOK

DELPHI 10.4 SYDNEY EDITION

Marco Cantù

Object Pascal Handbook Delphi 10.4 Sydney Edition

**The Complete Guide to the Object Pascal
programming language for Delphi 10.4 Sydney**

Original edition: Piacenza (Italy), July 2015

Delphi 10.4 Edition: Piacenza (Italy), November 2020 Draft

Author: Marco Cantù

Publisher: Marco Cantù

First edition editor: Peter W A Wood

Cover Designer: Fabrizio Schiavi (www.fsd.it)

Copyright 1995-2020 Marco Cantù, Piacenza, Italy. World rights reserved.

The author created example code in this publication expressly for the free use by its readers. Source code for this book is copyrighted freeware, distributed via a GitHub project listed in the book and on the book web site. The copyright prevents you from republishing the code in print or electronic media without permission. Readers are granted limited permission to use this code in their applications, as long as the code itself is not distributed, sold, or commercially exploited as a stand-alone product.

Aside from this specific exception concerning source code, no part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, in the original or in a translated language, including but not limited to photocopy, photograph, magnetic, or other record, without the prior agreement and written permission of the publisher.

Delphi is a trademark of Embarcadero Technologies (a division of Idera, Inc.). Other trademarks are of the respective owners, as referenced in the text. Whilst the author and publisher have made their best efforts to prepare this book, they make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accepts no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

Object Pascal Handbook for Delphi 10.4 Sydney

This PDF version is the November 30, 2020 Draft of the book

ISBN-10: to be assigned

ISBN-13: to be assigned

The electronic edition of this book has been licensed to Embarcadero Technologies Inc. It's also sold directly by the author. *Do not distribute the PDF version of this book without permission.* The printed edition is published through Kindle Direct Publishing and sold on several outlets.

More information at ***<http://www.marcocantu.com/objectpascal>***

begin

*To my family, Raffaella, Benny, and Jacopo,
with all of my love and a big thank you for all you do
to take my life ahead of my expectations*

Power and simplicity, expressiveness and readability, great for learning and for professional development alike, these are some of the traits of today's Object Pascal, a language with a long history, a lively present, and a brilliant future ahead.

Object Pascal is a multi-faceted language. It combines the power of object-oriented programming, advanced support for generic programming and dynamic constructs like attributes, but without removing support for more traditional style of procedural programming. A tool for all trades, with compilers and development tools embracing the mobile era. A language ready for the future, but with solid roots in the past.

What is the Object Pascal language for? From writing desktop apps to client-server applications, from massive web server modules to middle-ware, from office automation to apps for the latest phones and tablets, from industrial automation systems to Internet virtual phone networks... this is not what the language could be used for, but what it is *currently* used for today, in the real world.

The core of the Object Pascal language as we use today comes from its definition in 1995, a terrific year for programming languages, given that this was the year Java

4 - begin

and JavaScript were also invented. While the root of the language dates back to its Pascal ancestor, its evolution didn't stop in 1995, with core enhancements continuing as of today, with the desktop and mobile compilers build by Embarcadero Technologies and found in Delphi and RAD Studio.

A Book on Today's Language

Given the changing role of the language, its extension over the years, and the fact it is now attracting new developers, I felt it important to write a book that offers complete coverage of the Object Pascal language as it is today. The goal is to offer a language manual for new developers, for developers coming from other similar languages, but also for old timers of different Pascal dialects that want to learn more about recent language changes.

Newcomers certainly need some of the foundations, but given changes have been pervasive even old-timers will find something new from the initial chapters.

Beside a short Appendix covering the short history of the Object Pascal language, this book was written to cover the language as it is today. A large part of the core features of the language hasn't changed significantly since the early versions of the Delphi, the first implementation of modern Object Pascal in 1995.

As I'll hint throughout the book, the language has been far from stagnant during all of these years, it has been evolving at quite a fast pace. In other books I wrote in the past, I followed a more *chronological* approach, covering classic Pascal first, and following extensions more or less as they appeared over time. In this book, however, the idea is to use a more *logical* approach, progressing through the topics and covering how the language works today, and how to best use it, rather than how it evolved over time.

As an example, native data types dating back to the original Pascal language have method-like capabilities (thanks to intrinsic type helpers) introduced recently. So in Chapter 2 I'll introduce how to use this feature, although it won't be until much later than you'll figure out how to make such custom type extensions yourself.

In other words, this book covers the Object Pascal language as it is today, teaching it from the ground up, with only a very limited historical perspective. Even if you have used the language in the past, you might want to skim through the entire text looking for newer features, and not focus only on the final chapters.

Learn by Doing

The idea of the book is to explain core concepts and immediately present short demos that readers are encouraged to try to execute, experiment with, and extend to understand the concepts and assimilate them better. The book is not a reference manual, explaining what the language should do in theory and listing all possible corner cases. While trying to be precise, the focus is more on teaching the language offering a practical step-by-step guide. Examples are generally very simple, because the goal is to have them focused on one feature at a time.

The entire source code is available in an online code repository on GitHub. You can download it as a single file, clone the repository, or just browse it online and download only the code of specific projects. If you get from the repository, you'd easily update your code in case I publish any changes or additional demos. The location on GitHub is:

|| <https://github.com/MarcoDelphiBooks/ObjectPascalHandbook104>

To compile and test the demo code, you'll need a recent version of Delphi (at least 10.4 to run everything, but most of the demos will work on 10.x releases).

If you don't have a Delphi license there is a trial version available that you can use, generally allowing you 30-days free use of the compiler and IDE. There is also a free Delphi Community Edition (currently updated to the 10.3 version) that is free to use for anyone with no or limited earnings from software development work.

Acknowledgments

As any book, this volumes owes a lot to many people, too many to list one by one. The person who shared most of the work on the first edition of this book was my editor, Peter Wood, who kept bearing with my ever changing schedule and was able to improve my technical English, helping to make this book what it is.

After the first edition, a reader, Andreas Toth, sent me extensive feedback of the previous edition and his suggestions have helped improving the text. This new edition has also been reviewed by a few other Delphi experts (most of them among the Embarcadero MVPs) including, in particular, François Piette, who sent me dozens of corrections and suggestions.

Given my current work position as product manager at Embarcadero Technologies, I owe a lot to all my coworkers and the members of the R&D team, as during my time at the company my understanding of the product and its technology has further increased thanks to the insight I've gained in countless conversations,

6 - begin

meetings, and email threads. Given how hard it is to make sure everyone is mentioned, I won't really try, but only pick three people that for their role had a direct input on the first edition of this book: David I of Developer Relations, John Thomas (JT) who was heading RAD Product Management, and RAD Architect Allen Bauer.

More recently I've been working extensively with the two other product managers for RAD Studio, Sarina DuPont and David Millington, with our current *evangelist* Jim McKeeth, and with the current group of outstanding R&D architects.

Other people outside Embarcadero continued being important contacts and at times offering direct input, from many Italian Delphi experts to the countless customers, Embarcadero sales and technical partners, Delphi community members, MVPs and even developers using other languages and tools I keep meeting so often.

If there is one person in this group I've spent a lot of time with, prior to joining Embarcadero, this is Cary Jensen, with whom I've organized a few rounds of Delphi Developer Days in Europe and the US. And finally big thank you goes to my family for bearing with my travel schedule, nights of meetings, plus some extra book writing on weekends. Thanks again Lella, Benny, and Jacopo.

About Myself, the Author

I've spent most part of the past 25 years writing, teaching, and consulting on software development with the Object Pascal language. I wrote the Mastering Delphi best-selling series and later self-published several Handbooks on the development tool (about the different versions from Delphi 2007 to Delphi XE).

I have spoken at a large number of programming conferences in most continents, and taught to thousands of developers at conferences, Delphi developer events, classes hosted by companies, online webinars and CodeRage conferences.

Having worked as an independent consultant and trainer for many years, in 2013 my career took a sudden change: I accepted a position as Delphi and now RAD Studio product manager at Embarcadero Technologies, the company that builds and sells these development tools. To avoid annoying you any further, I'll only add that I currently live in Italy, commute to California (a bit less, recently), have a lovely wife and two wonderful kids, and enjoy getting back to programming as much as I can.

I hope you enjoy reading the book, as much as I enjoyed writing this new edition. For further information, use the following web sites and social media channels:

```
http://www.marcocantu.com/objectpascalhandbook
http://blog.marcocantu.com
http://twitter.com/marcocantu
```

table of contents

begin.....	3
A Book on Today's Language.....	4
Learn by Doing.....	5
Acknowledgments.....	5
About Myself, the Author.....	6
Table of Contents.....	7
Part I: Foundations.....	17
Summary of Part I.....	18
01: Coding in Pascal.....	19
Let's Start with Code.....	19
A First Console Application.....	20
A First Visual Application.....	21
Syntax and Coding Style.....	24
Comments.....	25
Comments and XML Doc.....	26
Symbolic Identifiers.....	28
White Space.....	30
Indentation.....	31
Syntax Highlighting.....	32
Language Keywords.....	33
The Structure of a Program.....	38
Unit and Program Names.....	39

8 - Table of Contents

Units and Scope.....	43
The Program File.....	45
Compiler Directives.....	46
Conditional Defines.....	46
Compiler Versions.....	47
Include Files.....	49
02: Variables and Data Types.....	51
Variables and Assignments.....	52
Literal Values.....	53
Assignment Statements.....	54
Assignments and Conversion.....	55
Initializing Global Variables.....	55
Initializing Local Variables.....	56
Inline Variables.....	56
Constants.....	58
Lifetime and Visibility of Variables.....	60
Data Types.....	61
Ordinal and Numeric Types.....	62
Boolean.....	67
Characters.....	67
Floating Point Types.....	70
Simple User-Defined Data Types.....	72
Named vs. Unnamed Types.....	73
Type Aliases.....	74
Subrange Types.....	75
Enumerated Types.....	76
Set Types.....	77
Expressions and Operators.....	79
Using Operators.....	79
Operators and Precedence.....	80
Date and Time.....	83
Typecasting and Type Conversions.....	85
03: Language Statements.....	89
Simple and Compound Statements.....	90
The If Statement.....	91
Case Statements.....	93
The For Loop.....	94
The for-in Loop.....	97
While and Repeat Statements.....	99
Examples of Loops.....	100
Breaking the Flow with Break and Continue.....	102
04: Procedures and Functions.....	105
Procedures and Functions.....	105
Forward Declarations.....	108

Table of Contents - 9

A Recursive Function.....	109
What Is a Method?.....	111
Parameters and Return Values.....	111
Exit with a Result.....	112
Reference Parameters.....	113
Constant Parameters.....	115
Function Overloading.....	116
Overloading and Ambiguous Calls.....	118
Default Parameters.....	119
Inlining.....	121
Advanced Features of Functions.....	124
Object Pascal Calling Conventions.....	124
Procedural Types.....	125
External Functions Declarations.....	127
o5: Arrays and Records.....	131
Array Data Types.....	132
Static Arrays.....	132
Array Size and Boundaries.....	133
Multi-Dimensional Static Arrays.....	134
Dynamic Arrays.....	136
Open Array Parameters.....	139
Record Data Types.....	143
Using Arrays of Records.....	145
Variant Records.....	146
Fields Alignments.....	147
What About the With Statement?.....	148
Records with Methods.....	150
Self: The Magic Behind Records.....	152
Initializing Records.....	153
Records and Constructors.....	154
Operators Gain New Ground.....	155
Operators and Custom Managed Record.....	159
Variants.....	164
Variants Have No Type.....	164
Variants in Depth.....	165
Variants Are Slow.....	166
What About Pointers?.....	168
File Types, Anyone?.....	171
o6: All About Strings.....	173
Unicode: An Alphabet for the Entire World.....	174
Characters from the Past: from ASCII to ISO Encodings.....	174
Unicode Code Points and Graphemes.....	175
From Code Points to Bytes (UTF).....	176
The Byte Order Mark.....	178

10 - Table of Contents

Looking at Unicode.....	179
The Char Type Revisited.....	182
Unicode Operations With The Character Unit.....	182
Unicode Character Literals.....	184
What about 1-Byte Chars?.....	186
The String Data Type.....	186
Passing Strings as Parameters.....	189
The Use of [] and String Characters Counting Modes.....	190
Concatenating Strings.....	192
The String Helper Operations.....	194
More String RTL Functions.....	197
Formatting Strings.....	198
The Internal Structure of Strings.....	200
Looking at Strings in Memory.....	202
Strings and Encoding.....	203
Other Types for Strings.....	206
The UCS4String type.....	207
Older String Types.....	207
Part II: OOP in Object Pascal.....	209
Summary of Part II.....	210
o7: Objects.....	211
Introducing Classes and Objects.....	212
The Definition of a Class.....	212
Classes in Other OOP Languages.....	214
The Class Methods.....	215
Creating an Object.....	215
The Object Reference Model.....	216
Disposing Objects.....	217
What is “Nil”?.....	218
Records vs. Classes in Memory.....	219
Private, Protected, and Public.....	219
An Example of Private Data.....	221
Encapsulation and Forms.....	223
The Self Keyword.....	225
Creating Components Dynamically.....	226
Constructors.....	228
Managing Local Class Data with Constructors and Destructors.....	230
Overloaded Methods and Constructors.....	231
The Complete TDate Class.....	233
Nested Types and Nested Constants.....	236
o8: Inheritance.....	239
Inheriting from Existing Types.....	240
A Common Base Class.....	242
Protected Fields and Encapsulation.....	243

Using the “Protected Hack”	244
From Inheritance to Polymorphism.....	246
Inheritance and Type Compatibility.....	246
Late Binding and Polymorphism.....	248
Overriding, Redefining, and Reintroducing Methods.....	250
Inheritance and Constructors.....	252
Virtual versus Dynamic Methods.....	253
Abstracting Methods and Classes.....	255
Abstract Methods.....	255
Sealed Classes and Final Methods.....	257
Safe Type Cast Operators.....	258
Visual Form Inheritance.....	260
Inheriting From a Base Form.....	261
09: Handling Exceptions.....	265
Try-Except Blocks.....	266
The Exceptions Hierarchy.....	268
Raising Exceptions.....	270
Exceptions and the Stack.....	271
The Finally Block.....	272
Restore the Cursor with a Finally Block.....	274
Restore the Cursor with a Managed Record.....	275
Exceptions in the Real World.....	276
Global Exceptions Handling.....	276
Exceptions and Constructors.....	278
Advanced Features of Exceptions.....	280
Nested Exceptions and the InnerException Mechanism.....	280
Intercepting an Exception.....	283
10: Properties and Events.....	285
Defining Properties.....	286
Properties Compared to Other Programming Languages.....	287
Code Completion for Properties.....	289
Adding Properties to Forms.....	290
Adding Properties to the TDate Class.....	292
Using Array Properties.....	294
Setting Properties by Reference.....	295
The published Access Specifier.....	296
Design-Time Properties.....	297
Published and Forms.....	298
Automatic RTTI.....	299
Event-Driven Programming.....	300
Method Pointers.....	301
The Concept of Delegation.....	303
Events Are Properties.....	305
Adding an Event to the TDate Class.....	307

12 - Table of Contents

Creating a TDate Component.....	309
Implementing Enumeration Support in a Class.....	311
15 Tips About Mixing RAD and OOP.....	314
Tip 1: A Form is a Class.....	315
Tip 2: Name Components.....	315
Tip 3: Name Events.....	315
Tip 4: Use Form Methods.....	316
Tip 5: Add Form Constructors.....	316
Tip 6: Avoid Global Variables.....	316
Tip 7: Never Use Form1 In TForm1 Methods.....	317
Tip 8: Seldom Use Form1 In Other Forms.....	317
Tip 9: Remove the Global Form1 Variable.....	317
Tip 10: Add Form Properties.....	318
Tip 11: Expose Components Properties.....	318
Tip 12: Use Array Properties When Needed.....	318
Tip 13: Starting Operations In Properties.....	319
Tip 14: Hide Components.....	319
Tip 15: Use an OOP Form Wizard.....	320
Tips Conclusion.....	320
11: Interfaces.....	321
Using Interfaces.....	322
Declaring an Interface.....	323
Implementing the Interface.....	324
Interfaces and Reference Counting.....	326
Errors in Mixing References.....	327
Weak And Unsafe Interface References.....	328
Advanced Interface Techniques.....	331
Interface Properties.....	331
Interface Delegation.....	332
Multiple Interfaces and Methods Aliases.....	334
Interface Polymorphism.....	336
Extracting Objects from Interface References.....	337
Implementing An Adapter Pattern with Interfaces.....	339
12: Manipulating Classes.....	343
Class Methods and Class Data.....	343
Class Data.....	344
Virtual Class Methods and the Hidden Self Parameter.....	345
Class Static Methods.....	345
Class Properties.....	348
A Class with an Instance Counter.....	348
Class Constructors (and Destructors).....	349
Class Constructors in the RTL.....	351
Implementing the Singleton Pattern.....	351
Class References.....	352

Class References in the RTL.....	354
Creating Components Using Class References.....	354
Class And Record Helpers.....	357
Class Helpers.....	357
Class Helpers and Inheritance.....	360
Adding a Controls Enumeration with a Class Helper.....	360
Record Helpers for Intrinsic Types.....	363
Helpers for Type Aliases.....	365
13: Objects and Memory.....	367
Global Data, Stack, and Heap.....	368
Global Memory.....	368
Stack.....	369
Heap.....	370
The Object Reference Model.....	371
Passing Objects as Parameters.....	371
Memory Management Tips.....	373
Destroying Objects You Create.....	373
Destroying Objects Only Once.....	374
Memory Management and Interfaces.....	376
More on Weak References.....	377
The Unsafe Attribute.....	380
Tracking and Checking Memory.....	381
Memory Status.....	381
FastMM4.....	382
Tracking Leaks and Other Global Settings.....	382
Buffer Overruns in the Full FastMM4.....	384
Memory Management on Platforms Other than Windows.....	386
Tracking Per-Class Allocations.....	387
Writing Robust Applications.....	387
Constructors, Destructors, and Exceptions.....	388
Nested Finally blocks.....	389
Dynamic Type Checking.....	390
Is this Pointer an Object Reference?.....	392
Part III: Advanced Features.....	395
Chapters of Part III.....	396
14: Generics.....	397
Generic Key-Value Pairs.....	398
Inline Variables and Generics Type Inference.....	401
Type Rules on Generics.....	401
Generics in Object Pascal.....	402
Generic Types Compatibility Rules.....	403
Generic Methods for Standard Classes.....	404
Generic Type Instantiation.....	406
Generic Type Functions.....	408

14 - Table of Contents

Class Constructors for Generic Classes.....	411
Generic Constraints.....	412
Class Constraints.....	413
Specific Class Constraints.....	414
Interface Constraints.....	415
Interface References vs. Generic Interface Constraints.....	417
Default Constructor Constraint.....	418
Constraints Summary and Combining Them.....	420
Predefined Generic Containers.....	421
Using TList<T>.....	421
Sorting a TList<T>.....	423
Sorting with an Anonymous Method.....	424
Object Containers.....	426
Using a Generic Dictionary.....	426
Dictionaries vs. String Lists.....	430
Generic Interfaces.....	431
Predefined Generic Interfaces.....	434
Smart Pointers in Object Pascal.....	435
Using Records for Smart Pointers.....	435
Implementing a Smart Pointer with a Generic Managed Record.....	436
Implementing a Smart Pointer with a Generic Record and an Interface.....	438
Adding Implicit Conversion.....	440
Comparing Smart Pointer Solutions.....	441
Covariant Return Types with Generics.....	442
Of Animals, Dogs, and Cats.....	442
A Method with a Generic Result.....	443
Returning a Derived Object of a Different Class.....	444

15: Anonymous Methods.....445

Syntax and Semantics of Anonymous Methods.....	446
An Anonymous Method Variable.....	447
An Anonymous Method Parameter.....	447
Using Local Variables.....	448
Extending the Lifetime of Local Variables.....	448
Anonymous Methods Behind the Scenes.....	450
The (Potentially) Missing Parenthesis.....	451
Anonymous Methods Implementation.....	452
Ready To Use Reference Types.....	453
Anonymous Methods in the Real World.....	454
Anonymous Event Handlers.....	454
Timing Anonymous Methods.....	456
Threads Synchronization.....	457
AJAX in Object Pascal.....	460

16: Reflection and Attributes.....465

Extended RTTI.....	466
--------------------	-----

Table of Contents - 15

A First Example.....	466
Compiler Generated Information.....	468
Weak and Strong Types Linking.....	469
The RTTI Unit.....	470
The RTTI Classes in the Rtti Unit.....	472
RTTI Objects Lifetime Management and the TRttiContext record.....	473
Displaying Class Information.....	475
RTTI for Packages.....	476
The TValue Structure.....	477
Reading a Property with TValue.....	479
Invoking Methods.....	480
Using Attributes.....	481
What is an Attribute?.....	481
Attribute Classes and Attribute Declarations.....	482
Browsing Attributes.....	484
Virtual Methods Interceptors.....	486
RTTI Case Studies.....	490
Attributes for ID and Description.....	490
XML Streaming.....	495
Other RTTI-Based Libraries.....	501
17: Object and the System Unit.....	503
The TObject Class.....	504
Construction and Destruction.....	504
Knowing About an Object.....	505
More Methods of the TObject Class.....	506
TObject's Virtual Methods.....	508
TObject Class Summary.....	511
Unicode and Class Names.....	512
The System Unit.....	513
Selected System Types.....	513
Interfaces in the System Unit.....	514
Selected System Routines.....	515
Predefined RTTI Attributes.....	516
18: Other Core RTL Classes.....	517
The Classes Unit.....	518
The Classes in the Classes Unit.....	518
The TPersistent Class.....	520
The TComponent Class.....	520
Modern File Access.....	523
The Input/Output Utilities Unit.....	523
Introducing Streams.....	525
Using Readers and Writers.....	527
Building Strings and String Lists.....	529
The TStringBuilder class.....	529

16 - Table of Contents

Using String Lists.....	531
The Run-Time Library is Quite Large.....	531
In Closing.....	535
end.....	537
Appendix Summary.....	537
A: The Evolution of Object Pascal.....	539
Wirth's Pascal.....	540
Turbo Pascal.....	540
The early days of Delphi's Object Pascal.....	541
Object Pascal From CodeGear to Embarcadero.....	542
Going Mobile.....	543
The Delphi 10.x Period.....	544
B: Glossary.....	545
A.....	545
B.....	546
C.....	547
D.....	548
E.....	549
F.....	549
G.....	550
H.....	550
I.....	551
M.....	552
O.....	552
P.....	553
R.....	553
S.....	555
U.....	555
V.....	556
W.....	556
C: Index.....	557

part i: foundations

Object Pascal is an extremely powerful language based upon core foundations such as a good program structure and extensible data types. These foundations are partially derived from the traditional Pascal language, but even the core language features have seen many extensions from the early days.

In this first part of the book, you'll learn about the language syntax, the coding style, the structure of programs, the use of variables and data types, the fundamental language statements (like conditions and loops), the use of procedures and functions, and core type constructors such as arrays, records, and strings.

These are the foundations of the more advanced features, from classes to generic types, that we'll explore in the second and third parts of the book. Learning a language is like building a house, and you need to start on solid ground and good foundations, or everything else up and above would be shining... but shaky.

Summary of Part I

Chapter 1: Coding in Pascal

Chapter 2: Variables and Data Types

Chapter 3: Language Statements

Chapter 4: Procedures and Functions

Chapter 5: Arrays and Records

Chapter 6: All About Strings

01: coding in pascal

This chapter starts with some of the building blocks of an Object Pascal application, covering standard ways of writing code and related comments, introducing keywords, and the structure of a program. I'll start writing some simple applications, trying to explain what they do and thus introducing some other key concepts covered in more details in the coming chapters.

Let's Start with Code

This chapter covers the foundation of the language, but it will take me a few chapters to guide you through the details of a complete working application. So for now let's have a look at two first programs (different in their structure), without really getting into too many details. Here I just want to show you the structure of programs that I'll use to build demos explaining specific language constructs before I'll

20 - 01: Coding in Pascal

be able to cover all of the various elements. Given that I want you to be able to start putting the information in the book into practice as soon as possible, looking at demo examples from the beginning would be a good idea.

Object Pascal has been designed to work hand-in-glove with its Integrated Development Environment. It is through this powerful combination that Object Pascal can match the ease of development speed of programmer-friendly languages and at the same time match the running speed of machine-friendly languages.

The IDE lets you design user interfaces, help you write code, run your programs and much, much more. I'll be using the IDE throughout this book as I introduce the Object Pascal language to you.

A First Console Application

As a starting point, I'm going to show you the code of a simple *Hello, World* console application showing some of the structural elements of an Object Pascal program. A console application is a program with no graphical user interface, displaying text and accepting keyboard input, and generally executed from an operating system console or command prompt. Console apps generally make little sense on mobile platforms, but they are still used on Windows (where Microsoft has recently spent effort on `cmd.exe` improvements, PowerShell, and terminal access) and fairly popular on Linux.

I won't explain what the different elements of the code below mean just yet, as that is the purpose of the first few chapters of the book. Here is the code, from the `HelloConsole` application project:

```
program HelloConsole;  
  
{$APPTYPE CONSOLE}  
  
var  
    StrMessage: string;  
  
begin  
    StrMessage := 'Hello, World';  
    writeln (StrMessage);  
    // wait until the Enter key is pressed  
    readln;  
end.
```

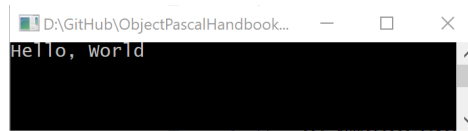
note As explained in the introduction, the complete source code of all of the demos covered in the book is available in an online repository on GitHub. Refer to the book introduction for more details on how to get those demos. In the text I refer to the project name (in this case `HelloConsole`), which is also the name of the folder containing the various files of the demo. The project folders are grouped by chapter, so you'll find this first demo under `01/HelloConsole`.

You can see the program name in the first line after a specific declaration, a compiler directive (prefixed by the `$` symbol and enclosed in curly braces), a variable declaration (a string with a given name), and three lines of code plus a comment within the main `begin-end` block. Those three lines of code copy a value into the string, call a system function to write that line of text to the console, and call another system function to read a line of user input (or in this case to wait until the user pressed the Enter key). As we'll see, you can define your own functions, but Object Pascal comes with hundreds of predefined ones.

Again, we'll learn about all of these elements soon, as this initial section serves only to give you an idea of what a small but complete Pascal program looks like. Of course you can open and run this application, which will produce output like the following (the actual Windows version is displayed in Figure 1.1).

■ *Hello, world*

Figure 1.1:
The output of the
HelloConsole example,
running on Windows



A First Visual Application

A modern application, though, rarely looks like this old fashioned console program, but is generally made of visual elements (referred to as controls) displayed in windows (referred to as forms). In most cases in this book I'll build visual demos (even if in most cases they'll boil down to displaying simple text) using the FireMonkey library (which is also known as FMX).

note In Delphi the visual controls come in two *flavors*: VCL (Visual Component Library for Windows) and FireMonkey (a multi-device library for all supported platforms, desktop and mobile). In any case, it should be rather simple to adapt the demos to the Windows-specific VCL library.

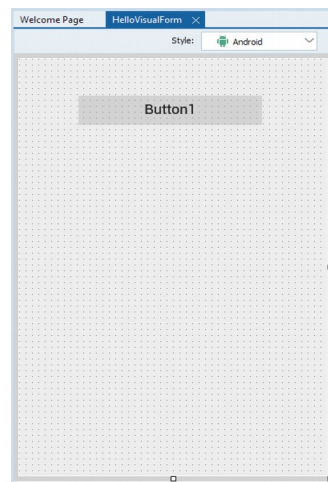
22 - 01: Coding in Pascal

To understand the exact structure of a visual application, you'll have to read a good part of this book, as a form is an object of a given class and has methods, event handlers, and properties... all features that will take a while to go through. But to be able to create these applications, you don't need to be an expert, as all you have to do is use a menu command to create a new desktop or mobile application. What I'll do in the initial part of the book is to base the demos on the FireMonkey platform and simply use the context of forms and button click operations. To get started, you can create a form of any type (desktop or mobile, I'd generally pick a multi-device “blank” application, as it will also run on Windows), and place a button control on it, with a multi-line text control (or Memo) below it to display the output.

Figure 1.2 shows how your application form will look for a mobile application in the Delphi IDE, after selecting an Android style preview (see the combo box above the design surface) and adding a single control, a button.

Figure 1.2:

A simple mobile application with a single button, used by the HelloVisual demo



What you have to do to create a similar application is to add a button to an empty form. Now to add the actual code, which is the only thing we are interested in for now, double click on the button, you'll be presented with the following code skeleton (or something very similar):

```
procedure TForm1.Button1Click (Sender: TObject)
begin
    ;
end;
```

Even if you don't know what a method of a class is (which is what `Button1Click` is), you can type something in that code fragment (that means within the `begin` and `end` keywords) and that code will execute when you press the button.

Our first “visual” program has code matching that of the first console application, only in a different context and calling a different library function, namely the `ShowMessage` global function used to display some a string in a message box. This is the code you can find in the `HelloVisual` application project and you can try rebuilding it from scratch quite easily:

```
procedure TForm1.Button1Click (Sender: TObject)
var
    StrMessage: string;
begin
    StrMessage := 'Hello, world';
    ShowMessage (StrMessage);
end;
```

Notice how you need to place the declaration of the `strMessage` variable before the `begin` statement and the actual code after it. Again, don't worry if things are not clear, everything will get explained in due time and in great detail.

note You can find the source code of this demo in a folder under the 01 container for the chapter. In this case, however, there is a project file name like the demo but also a secondary unit file with the word “Form” added after the project name. That's the standard I'm going to follow in the book. The structure of a project is covered at the end of this chapter.

In Figure 1.3 you can see the output of this simple program, running on Windows with FMX MobilePreview mode enabled (you can run this demo on Android, iOS, and macOS as well, but that requires some extra configurations in the IDE).

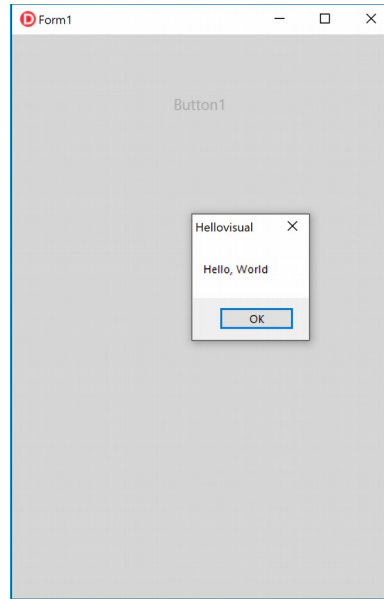
note The FireMonkey Mobile Preview makes a Windows application look a bit like a mobile app. I've enable this mode in most of the demos of this book. This is done by adding a `uses` statement for the `MobilePreview` unit in the project source code.

Now that we have a way to write and test a demo program, let's get back to square one, by covering all of the details of the first few building blocks of an application, as I promised at the beginning of this chapter. The first thing you need to know is how to *read* a program, how the various elements are written, and what is the structure of the application we just build (which has both a PAS file and DPR file).

24 - 01: Coding in Pascal

Figure 1.3:

A simple mobile application with a single button, used by the HelloVisual demo



Syntax and Coding Style

Before we move on to the subject of writing actual Object Pascal language statements, it is important to highlight some elements of Object Pascal coding style. The question I'm addressing here is this: Besides the syntax rules (which we still haven't looked into), how should you write code? There isn't a single answer to this question, since personal taste can dictate different styles. However, there are some principles you need to know regarding comments, uppercase, spaces, and what many years ago was called *pretty-printing* (pretty for us human beings, not the computer), a term now considered obsolete.

In general, the goal of any coding style is clarity. The style and formatting decisions you make are a form of shorthand, indicating the purpose of a given piece of code. An essential tool for clarity is consistency—whatever style you choose, be sure to follow it throughout a project and across projects.

tip The IDE (Integrated Development Environment) has support for automatic code formatting (at the unit or project level): You can ask the editor to re-format your code with the Ctrl+D keys, following a set of rules you can change by tweaking about 40 different formatting elements (found among the IDE Options), and even share these settings with other developers on your team to make formatting consistent. However, the automatic formatting does not support some of the most recent language features.

Comments

Although code is often self-explanatory, it is relevant to add a significant amount of comments in the source code of a program, to further explain to others (and to yourself when you look at your code a long time in the future) why the code was written in a given way and what were the assumptions.

In traditional Pascal comments were enclosed in either braces or parentheses followed by a star. Modern versions of the language also accept the C++ style single-line comments, double slash, which span to the end of the line and require no symbol to indicate the end the comment:

```
// this is a comment up to the end of the line  
{ this is a multiple lines comment }  
(* this is another  
multiple lines comment *)
```

The first form of comment is by far the most common but it wasn't originally part of Pascal. It has been borrowed from C/C++, which also use the `/* comment */` syntax for multiples lines comments, along with C#, Objective-C, Java, and JavaScript.

The second form is more commonly used than the third, which was often preferred in Europe because many European keyboards lack the brace symbol (or make it difficult to use, though a multiple keys combination). In other words, the oldest syntax is a bit out of fashion.

Comments up to the end of the line are very helpful for short comments and for commenting out a single line of code. They are by far the most common form of comments in the modern Object Pascal language.

tip In the IDE editor, you can comment or uncomment the current line (or a group of selected lines) with a direct keystroke. This is Ctrl+/ on the US keyboard and a different combination (with the physical / key) on other keyboards: The actual key is listed in the pop-up menu of the editor.

26 - 01: Coding in Pascal

Having three different forms of comments can be helpful for marking nested comments. If you want to comment out several lines of source code to disable them, and these lines contain some real comments, you cannot use the same comment identifier:

```
{
  code...
  {nested comment, creating problems}
  code...
}
```

The code above results in a compiler error, as the first closed brace indicates the end of the entire commented section. With a second comment identifier, you can write the following code, which is correct:

```
{
  code...
  // this comment is OK
  code...
}
```

An alternative is to comment out a group of lines as explained above, given it will add a second `//` comment to the commented line, you can easily remove by uncommenting the same block (preserving the original comment).

note If the open brace or parenthesis-star is followed by the dollar sign(\$), it is not a comment any more, but it becomes a compiler directive, as we have seen in the first demo in the line `{ $APPTYPE CONSOLE }`. Compiler directives instruct the compiler to do something special, and are briefly explained towards the end of this chapter.

Actually, compiler directives are still comments. For example, `{ $X+ This is a comment }` is legal. It's both a valid directive and a comment, although most *sane* programmers will probably tend to separate directives and comments.

Comments and XML Doc

There is a special version of comments, common also to other programming languages, that is treated in a particular way by the compiler. These special comments generate additional documentation available directly to the IDE Help Insight and in XML files generated by the compiler.

note In the Delphi IDE, Help insight automatically displays information about a symbol (including its type and where it has been defined). With XML Doc comments you can augment this information with specific details written in the source code itself.

XML Doc is enabled by using `///` comments or `{!}` comments. Within these comments you can use general text or (better) specific XML tags to indicate information about the commented symbol, its parameters and return value and more. This is a very simple case of free form text:

```
public
  /// This is a custom method, of course
  procedure CustomMethod;
```

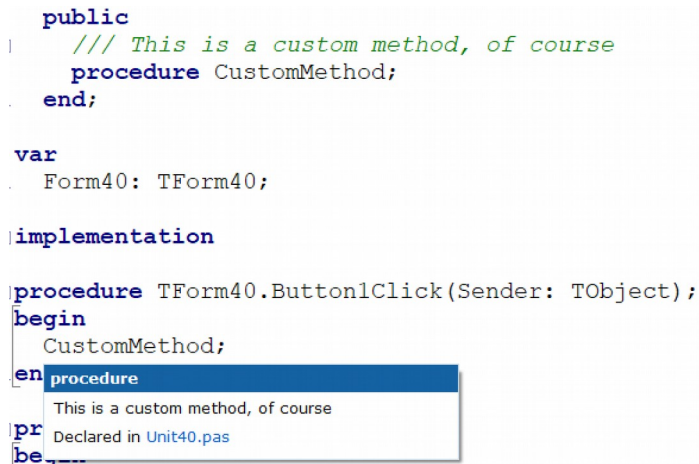
The information is added to the XML output generated by the compiler if you enable XML Doc generation, as follows:

```
<procedure name="CustomMethod" visibility="public">
  <devnotes>
    This is a custom method, of course
  </devnotes>
</procedure>
```

The same information is displayed in the IDE as you hover over the symbol, as shown in Figure 1.4.

Figure 1.4:

Help Insight in the Delphi IDE displays XML Doc information written in `///` comments



```
public
  /// This is a custom method, of course
  procedure CustomMethod;
end;

var
  Form40: TForm40;

implementation

procedure TForm40.Button1Click(Sender: TObject);
begin
  CustomMethod;
end
procedure
  This is a custom method, of course
pr
be
```

If you provide a summary section within the comment, following the recommended guidelines, this will equally show up in the Help Insight window:

```
public
  /// <summary>This is a custom method, of course</summary>
  procedure CustomMethod;
```

The advantage is that there are many other XML tags you can use for parameters, return values, and more detailed information. The available tags are listed at:

http://docwiki.embarcadero.com/RADStudio/en/XML_Documentation_Comments

Symbolic Identifiers

A program is made of many different symbols you can introduce to name the various elements (data types, variables, functions, objects, classes, and so on). Although you can use almost any identifier you want, there are a few rules you have to follow:

- Identifiers cannot include spaces (as spaces do separate identifiers from other language elements)
- Identifiers can use letters and numbers, including the letters in the entire Unicode alphabet; so you can name symbols in your own language if you want (something not really recommended, as some of the tools part of the IDE might not offer the same support)
- Out of the traditional ASCII symbols, identifiers can use only the underscore symbol (`_`); all other ASCII symbols beside letters and numbers are not allowed. Illegal symbols in identifiers include match symbols (`+`, `-`, `*`, `/`, `=`), all parenthesis and braces, punctuation, special characters (including `@`, `#`, `$`, `%`, `^`, `&`, `\`, `|`). What you can use, though, are Unicode symbols, like ☀️ or ∞.
- Identifiers must start with a letter or the underscore, starting with a number is not allowed (in other words, you can use numbers, but not as the first symbol). Here with numbers we refer to the ASCII numbers, 0 to 9, while other Unicode representations of numbers are allowed.

The following are examples of classic identifiers, listed in the `IdentifiersTest` application:

```
MyValue
Value1
My_Value
_Value
Val123
_123
```

These are example of legal Unicode identifiers (where the last is a bit extreme):

```
Cantù (Latin accented letter)
结 (Cash Balance in Simplified Chinese)
画像 (picture in Japanese)
☀️ (Sun Unicode symbol)
```

These are a few examples of *invalid* identifiers:

```
123
1Value
My Value
My-Value
My%Value
```

tip In case you want to check for a valid identifier at runtime (something rarely needed, unless you are writing a tool to help other developers), there is a function in the runtime library that you can use, called `IsValidIdent`.

Case Insensitivity and the Use of Uppercase

Unlike many other languages, including all those based on the C syntax (like C++, Java, C#, and JavaScript), the Object Pascal compiler ignores the case, or capitalization, of the identifiers. Therefore, the identifiers `Myname`, `MyName`, `myname`, `myName`, and `MYNAME` are all exactly the same. In my personal opinion, case-insensitivity is definitely a positive feature, as syntax errors and other subtle mistakes can be caused by incorrect capitalization in case-sensitive languages.

If you consider the fact that you can use Unicode for identifiers, however, things get a bit more complicated, as the uppercase version of a letter is treated like the same element, while an accented version of the same letter is treated like a separate symbol. In other words:

```
cantu: Integer;
Cantu: Integer; // error: duplicate identifier
cantù: Integer; // correct: different identifier
```

warning There is only one exception to the case-insensitivity rule of the language: the *Register* procedure of a components' package must start with the uppercase *R*, because of a C++ compatibility issue. Of course, when you refer to identifiers exported by other languages (like a native operating system function) you might have to use the proper capitalization.

There are a couple of subtle drawbacks, however. First, you must be aware that these identifiers really are the same, so you must avoid using them as different elements. Second, you should try to be consistent in the use of uppercase letters, to improve the readability of the code.

A consistent use of case isn't enforced by the compiler, but it is a good habit to get into. A common approach is to capitalize only the first letter of each identifier. When an identifier is made up of several consecutive words (you cannot insert a space in an identifier), every first letter of a word should be capitalized:

```
MyLongIdentifier
MyVeryLongAndAlmostStupidIdentifier
```

This is often called “Pascal-casing”, to contrast it with the so-called “Camel-casing” of Java and other languages based on the C syntax, which capitalizes internal words with an initial lowercase letter, as in

```
myLongIdentifier
```

30 - 01: Coding in Pascal

Actually, it is more and more common to see Object Pascal code in which local variables use camel-casing (lowercase initial), while class elements, parameters and other more global elements use the Pascal-casing. In any case, in the book source code snippets I've tried to use Pascal-casing consistently for all symbols.

White Space

Other elements completely ignored by the compiler are the spaces, new lines, and tabs you add to the source code. All these elements are collectively known as *white space*. White space is used only to improve code readability; it does not affect the compilation in any way.

Unlike traditional BASIC, Object Pascal allows you to write a statement over several lines of code, splitting a long instruction over two or more lines. The drawback of allowing statements over more than one line is that you have to remember to add a semicolon to indicate the end of a statement, or more precisely, to separate one statement from the next. The only restriction in splitting programming statements on different lines is that a string literal may not span several lines.

Although odd, the following lines all represent the same statement:

```
A := B + 10;

A :=
  B
  +
  10;

A
:=
// this is a mid-statement comment
B + 10;
```

Again, there are no fixed rules on the use of spaces and multiple-line statements, just some rules of thumb:

- The editor has a vertical line you can place after 80 or so characters. If you use this line and try to avoid surpassing this limit, your source code will look better and you won't have to scroll horizontally to read it on a computer with a smaller screen. The original idea behind the 80 characters was to make the code look nicer when printed, something not so common these days (but still valuable).
- When a function or procedure has several complex parameters, it is common practice to place the parameters on different lines.

- You can leave a line completely white (blank) before a comment or to divide a long piece of code in smaller portions. Even this simple idea can improve the readability of the code.
- Use spaces to separate the parameters of a function call, and maybe even a space before the initial open parenthesis. Also I like keeping operands of an expression separated, although this is a matter of preference..

Indentation

The last suggestion on the use of white spaces relates to the typical Pascal language-formatting style and indentation.

note Indentation rules are subject to personal taste and I don't want to enter a *tabs-versus-spaces* battle. Here I'm just indicting that is the “most common” or “standard” formatting style in the Object Pascal world, which is the style used in Delphi libraries source code. Historically in the Pascal world this set of rules was indicated as pretty-printing, a term now fairly uncommon.

This rule is simple: Each time you need to write a compound statement, indent it two spaces (not a tab, like a C programmer would generally do) to the right of the current statement. A compound statement inside another compound statement is indented four spaces, and so on:

```
if ... then
    statement;

if ... then
begin
    statement1;
    statement2;
end;

if ... then
begin
    if ... then
        statement1;
        statement2;
    end;
end;
```

Again, programmers have different interpretations of this general rule. Some programmers indent the `begin` and `end` statements to the level of the inner code, other programmers put the `begin` at the end of the line of previous statement (in a C-like fashion). This is mostly a matter of personal taste.

A similar indentation format is often used for lists of variables or data types after the `type` and `var` keywords:

32 - 01: Coding in Pascal

```
type
  Letters = ( 'A', 'B', 'C' );
  AnotherType = ...

var
  Name: string;
  I: Integer;
```

note In the code above you might be wondering why two different types, string and Integer, are written with a different case for the initial letter. As the original Object Pascal Styles Guide stated, “*types such as Integer are just identifiers and appear with a first cap; strings, however, are declared with the reserved word string, which should be all lowercase.*”

In the past it was also common to use a column-based indentation of the separator, when declaring custom types and variables. In such a case, the code above will look like below:

```
type
  Letters      = ( 'A', 'B', 'C' );
  AnotherType = ...

var
  Name : string;
  I    : Integer;
```

Indentation is also used commonly for statements that continue from the previous line or for the parameters of a function (if you don't put each parameter on a separate line):

```
MessageDlg ( 'This is a message',
             mtInformation, [mbOk], 0 );
```

Syntax Highlighting

To make it easier to read and write Object Pascal code, the IDE editor has a feature called syntax highlighting. Depending on the meaning in the language of the words you type, they are displayed using different colors and font styles. By default, keywords are in bold, strings and comments are in color (and often in italic), and so on.

Reserved words, comments, and strings are probably the three elements that benefit most from this feature. You can see at a glance a misspelled keyword, a string not properly terminated, and the length of a multiple-line comment.

You can easily customize the syntax highlight settings using the Editor Colors page of the Options dialog box of the IDE. If you are the only person using your computer to look to Object Pascal source code, choose the colors you like. If you work closely

with other programmers, you should all agree on a standard color scheme. I often found that working on a computer with a different syntax coloring than the one I normally use was really confusing.

Error Insight and Code Insights

The IDE editor has many more features to help you write correct code. The most obvious is Error Insight, that places a red squiggle under source code elements it doesn't understand, in the same fashion that a word processor marks spelling mistakes.

note At times you need to compile your program a first time to avoid having Error Insight indications for perfectly legitimate code. Also saving a file such as a form might force the inclusion of the proper units required for the current components, solving incorrect Error Insight indications. These issues have been largely addressed by the new LSD-based (Language Server Protocol based) Code Insight in Delphi 10.4.

Other features, like Code Completion, help you write code by providing a list of legitimate symbols in the place where you are writing. When a function or method has parameters, you'll see those listed as you type. And you can also hover over a symbol to see its definition. However, these are editor specific features that I don't want to delve into in detail, as I want to remain focused on the language and not discuss the IDE editor in detail (even if it is by far the most common tools used for writing Object Pascal code).

Language Keywords

Keywords are all the identifiers reserved by the language. These are symbols that have a predefined meaning and role and you cannot use them in a different context. Formally there is a distinction between reserved words and directives: Reserved words cannot be used as identifiers, while directives have a special meaning but could be used also in a different context (although you are recommended not to do so). In practice, you should not use any keyword as an identifier.

If you write some code like the following (where `property` is indeed a keyword):

```
var
  property: string
```

you'll see an error message like:

34 - 01: Coding in Pascal

| E2029 Identifier expected but 'PROPERTY' found

In general when you misuse a keyword, you'll get different error messages depending on the situation, as the compiler recognizes the keyword, but gets confused by its position in the code or by the following elements.

Here I don't want to show you a complete list of keywords, as some of them are rather obscure and rarely used, but only list a few, grouping them by their role. It will take me several chapters to explore all of these keywords and others I'm skipping in this list. You can find the official reference at:

| [http://docwiki.embarcadero.com/RADStudio/en/Fundamental_Syntactic_Elements_\(Delphi\)#Reserved_words](http://docwiki.embarcadero.com/RADStudio/en/Fundamental_Syntactic_Elements_(Delphi)#Reserved_words)

note Notice that some keywords can be used in different contexts, and here I'm generally referring only to the most common context (although a couple of keywords are listed twice). One of the reasons is that over the years the compiler team wanted to avoid introducing new keywords, as this might break existing applications, so they *recycled* some of the existing ones.

So let's start our exploration of keywords with some you've already seen in the initial demo source code and that are used to define the **structure of an application project**:

program	Indicates the name of an application project
library	Indicates the name of a library project
package	Indicates the name of a package library project
unit	Indicates the name of a unit, a source code file
uses	Refers to other units the code relies upon
interface	The part of a unit with declarations
implementation	The part of a unit with the actual code
initialization	Code executed when a program starts
finalization	Code executed on program termination
begin	The start of a block of code
end	The end of a block of code

Another set of keywords relates to the declaration of different basic **data types and variables** of such data types:

<code>type</code>	Introduces a block of type declarations
<code>var</code>	Introduces a block of variable declarations
<code>const</code>	Introduces a block of constant declarations
<code>set</code>	Defines a <i>power set</i> data type
<code>string</code>	Defines a string variable or custom string type
<code>array</code>	Defines an array type
<code>record</code>	Defines a record type
<code>integer</code>	Defines an integer variable
<code>real, single, double</code>	Define floating point variables
<code>file</code>	Defines a file

note There are many other data types defined in Object Pascal that I will cover later.

A third group includes keywords is used for the **basic language statements**, such a condition and loops, including also functions and procedures:

<code>if</code>	Introduces a conditional statement
<code>then</code>	Separates the condition from the code to execute
<code>else</code>	Indicates possible alternative code
<code>case</code>	Introduces a conditional statement with multiple options
<code>of</code>	Separates the condition from the options
<code>for</code>	Introduces a fixes repetitive cycle
<code>to</code>	Indicates the final upper value of the <code>for</code> cycle
<code>downto</code>	Indicates the final lower value of the <code>for</code> cycle
<code>in</code>	Indicates the collection to iterate over in a cycle
<code>while</code>	Introduces a conditional repetitive cycle
<code>do</code>	Separates the cycle condition from the code
<code>repeat</code>	Introduces a repetitive cycle with a final condition
<code>until</code>	Indicates the final condition of the cycle

36 - 01: Coding in Pascal

with	Indicates a data structure to work with
function	A sub-routine or group of statements returning a result
procedure	A sub-routine or group of statements which doesn't return a result
inline	Requests the compiler to replace a function call with the actual code of the function, for faster execution
overload	Allows the reuse of the name of a function or procedure

Many other keywords relate with **classes and objects**:

class	Indicates a class type
object	Used to indicate an older class type (now deprecated)
abstract	A class that is not fully defined
sealed	A class from which other classes cannot inherit
interface	Indicates an interface type (listed also in the first group)
constructor	An object or class initialization method
destructor	An object or class cleanup method
virtual	A virtual method
override	The modified version of a virtual method
inherited	Refers to a method of the base class
private	Portion of a class or record not accessible from the outside
protected	Portion of a class with limited access from the outside
public	Portion of a class or record fully accessible from the outside
published	Portion of a class made specifically available to users

<code>strict</code>	A stronger limitation for private and protected sections
<code>property</code>	A symbol mapped to a value or method
<code>read</code>	The mapper for getting the value of a property
<code>write</code>	The mapper for setting the value of a property
<code>nil</code>	The value of a <i>zero</i> object (used also for other entities)

A smaller group of keywords is used for **exceptions handling** (see Chapter 11):

<code>try</code>	The start of an exception handling block
<code>finally</code>	Introduces code to be executed regardless of an exception
<code>except</code>	Introduces code to be executed in case of an exception
<code>raise</code>	Used to trigger an exception

Another group of keywords is used for **operators** and is covered in the section “Expressions and Operators” later in this chapter, (beside some advanced operators covered only in later chapters):

<code>as</code>	<code>and</code>	<code>div</code>
<code>is</code>	<code>in</code>	<code>mod</code>
<code>not</code>	<code>or</code>	<code>shl</code>
<code>shr</code>	<code>xor</code>	

Finally, here is partial list of other **less commonly used keywords**, including a few old ones you should really avoid using. Again, here is just a quick indication of where they are used, as many or covered later in the book:

<code>default</code>	Indicates the default value of a property
<code>dynamic</code>	A virtual method with a different implementation
<code>export</code>	Legacy keyword used for exporting, replaced by the one below
<code>exports</code>	In a DLL project, lists the functions to export
<code>external</code>	Refers to an external DLL function you want to bind to

38 - 01: Coding in Pascal

<code>file</code>	Used for the legacy <i>file</i> type, rarely used these days.
<code>forward</code>	Indicates a forward function declaration
<code>goto</code>	Jumps to a label in a specific location of the code. It is highly recommended to avoid <code>goto</code> .
<code>index</code>	Used for indexed properties and (rarely) when importing or exporting functions
<code>label</code>	Defines a label a <code>goto</code> statement jumps to. It is highly recommended to avoid <code>goto</code> .
<code>message</code>	An alternative keyword for virtual functions, associated with platform messages
<code>name</code>	Used for mapping external functions
<code>nodefault</code>	Indicates the properties has no default value
<code>on</code>	Used to trigger an exception
<code>out</code>	An alternative to <code>var</code> , indicating a parameter passed by reference but not initialized
<code>packed</code>	Changes the memory layout of a record or data structure
<code>reintroduce</code>	Allows reusing the name of a virtual function
<code>requires</code>	In a package, indicates dependent packages

Notice that the list of Object Pascal language keywords has seen very few additions over recent years, as any additional keyword implies potentially introducing compilation errors into some existing programs preventing that had happened to use one of the new keyword as a symbol. Most of the recent additions to the language required no new keyword, like generics and anonymous methods.

The Structure of a Program

You hardly ever write all of the code in a single file, although this was the case with the first simple console application I showed earlier in this chapter. As soon as you create a visual application, you get at least one secondary source code file beside the project file. This secondary file is called *unit* and it's indicated by the `PAS` extension

(for Pascal source unit), while the main project file uses the DPR extension (for Delphi Project file). Both files contain Object Pascal source code.

Object Pascal makes extensive use of units, or program modules. Units, in fact, can be used to provide modularity and encapsulation even without using objects and they act as namespaces. An Object Pascal application is generally made up of several units, including units hosting forms and data modules. In fact, when you add a new form to a project, the IDE actually adds a new unit, which defines the code of the new form.

Units do not need to define forms; they can simply define and make available a collection of routines, or one of more data types (including classes). If you add a new blank unit to a project, it will only contain the keywords used to delimit the sections a unit is divided into:

```
unit Unit1;

interface

implementation

end.
```

The structure of a simple unit, shown above, includes the following elements:

- First, a unit has a unique name corresponding to its file name (that is, the sample unit above must be stored in the file *Unit1.pas*).
- Second, the unit has an `interface` section declaring what is visible to other units.
- Third, the unit has an `implementation` section with implementation details, the actual code, and possibly other local declarations, not visible outside of the unit.

Unit and Program Names

As I mentioned a unit name must correspond to the name of the file of that unit.

The same is true for a program. To rename a unit, you should use the Rename option in the Project manager and the two will be kept in synch (while doing a Save As operation in the IDE you still keep unit name and file name in sync but you end up with two copies of the file on disk). Of course, you can also change the file name at the file system level, but if you don't also change the declaration at the beginning of the unit, you'll see an error when the unit is compiled (or even when it is loaded in the IDE). This is a sample error message you'll get if you change the declaration of a unit without updating also the file name:

```
■ [DCC Error] E1038 Unit identifier 'Unit3' does not match file name
```


40 - 01: Coding in Pascal

The implication of this rule is that a unit or program name must be a valid Pascal identifier, but also a valid file name in the file system. For example, it cannot contain a space, not special characters beside the underscore (`_`), as covered earlier in this chapter in the section on identifiers. Given units and programs must be named using an Object Pascal identifier, they automatically result in valid file names, so you should not worry about that. The exception, of course, would be using Unicode symbols that are not valid file names at the file system level.

Dotted Unit Names

There is an extension to the basic rules for unit identifiers: a unit name can use a dotted notation. So all of the following are all valid unit names:

```
unit1  
myproject.unit1  
mycompany.myproject.unit1
```

Following the general rules, these units need to be saved in files with the same dotted name (that is, a unit called `MyProject.Unit1` must be stored in the file *MyProject.Unit1.pas*).

The reason for this extension is that unit names must be unique, and with more and more units being provided by Embarcadero and by third party vendors, this became more and more complex. All of the RTL units and the various other units that ship as part of the product libraries now follow the dotted unit name rule, with specific prefixes denoting the area, such as:

- `System` for core RTL
- `Data` for database access and the like
- `FMX` for the FireMonkey platform, the single-source multi-device architecture for desktop and mobile
- `VCL` for the Visual Component Library for Windows

note You'd generally refer to a dotted unit names, including the library units, with the complete name. It is also possible to use only the last portion of the name in a reference (allowing backward compatibility with older code) by setting up a corresponding rule in the project options. This setting is called "Unit scope names" and it is a semicolon separated list. Notice, however, that using this feature slows down the compilation compared to using fully qualified unit names.

More on the Structure of a Unit

Beside the interface and implementation sections, a unit can have an optional `initialization` section with some start up code, to be executed when the program is

first loaded into memory. If there is an `initialization` section, you can also have a `finalization` section, to be executed on program termination.

note You can also add initialization code in a class constructor, a recent language feature covered in Chapter 12. Using class constructors helps the linker remove unneeded code, which is why it is recommended to use class constructors and class destructors, rather than the old initialization and finalization sections. As a historical note, the compiler still supports using the `begin` keyword in place of the `initialization` keyword. A similar use of `begin` is still standard in the project source code.

In other word, the general structure of a unit, with all its possible sections and some sample elements, is like the following:

```
unit unitName;

interface

  // other units we refer to in the interface section
  uses
    unitA, unitB, unitC;

  // exported type definitions
  type
    newType = TypeDefinition;

  // exported constants
  const
    Zero = 0;

  // global variables
  var
    Total: Integer;

  // list of exported functions and procedures
  procedure MyProc;

implementation

  // other units we refer to in the implementation
  uses
    unitD, unitE;

  // hidden global variable
  var
    PartialTotal: Integer;

  // all the exported functions must be coded
  procedure MyProc;
  begin
    // ... code of procedure MyProc
  end;
```

42 - 01: Coding in Pascal

```
initialization  
  // optional initialization code  
  
finalization  
  // optional clean-up code  
  
end.
```

The purpose of the interface part of a unit is to make details of what the unit contains and can be used by the main programs and by other units that will make use of the unit. Whereas the implementation part contains the nuts and bolts of the unit which are hidden from outside viewers. This is how Object Pascal can provide this so called encapsulation even without using classes and objects.

As you can see, the `interface` of a unit can declare a number of different elements, including procedures, functions, global variables, and data types. Data types are generally used the most. The IDE automatically places a new class data type in a unit each time you create a visual form. However, containing form definitions is certainly not the only use for units in Object Pascal. You can have code only units, with functions and procedures (in a traditional way) and with classes that do not refer to forms or other visual elements.

Within an interface or an implementation section, the declarations for types, variables, constants, and the like can be written in any order and can be repeated multiple times. You can have a few constants, some types, then more constants, other variables, and another types section. The only rule is that to refer to a symbol this needs to be declared before it is referenced, which is the reason you often need to have multiple sections.

The Uses Clause

The `uses` clause at the beginning of the interface section indicates which other units we need to access in the interface portion of the unit. This includes the units that define the data types we refer to in the definition of data types of this unit, such as the components used within a form we are defining.

The second `uses` clause, at the beginning of the implementation section, indicates additional units we need to access only in the implementation code. When you need to refer to other units from the code of the routines and methods, you should add elements in this second `uses` clause instead of the first, as this reduces dependencies and increases compilation time. All the units you refer to must be present in the project directory or in a directory of the search path.

tip You can set the Search Path for a project in the Project Options. The system also considers units in the Library path, which is a global setting of the IDE.

C++ programmers should be aware that the `uses` statement does not correspond to an include directive. The effect of a `uses` statement is to import just the pre-compiled interface portion of the units listed. The implementation portion of the unit is considered only when that unit is compiled. The units you refer to can be both in source code format (PAS) or compiled format (DCU).

Although seldom used, Object Pascal had also an `$INCLUDE` compiler directive that works similarly to C/C++ `includes`. These special include files are used by some libraries for sharing compiler directives or other settings among multiple units, and generally have the INC file extension. This directive is covered shortly at the end of this chapter.

warning Notice that compiled units in Object Pascal are compatible only if they are build with the same version of the compiler and system libraries. A unit compiled in an older version of the product is generally not compatible with a later version of the compiler. Updates that are part of the same release, maintain compatibility. In other words, a unit built in version 10.3.1 is compatible with all 10.3.x versions, but not with 10.2 or 10.4 versions.

Units and Scope

In Object Pascal units are the key to encapsulation and visibility and, in that sense, they are probably even more important than the `private` and `public` keywords of a class. The scope of an identifier (such as a variable, procedure, function, or a data type) is the portion of the code in which the identifier is accessible or *visible*. The basic rule is that an identifier is meaningful only within its scope—that is, only within the unit, function, or procedure in which it is declared. You cannot use an identifier outside its scope.

note Until recently, Object Pascal didn't have the concept of scope limited to a generic code block that can include a declaration. Starting with Delphi 10.3 you can declare inline variables within a `begin-end` block to limit the scope of the variable to the specific block, like it happens in C or C++. More details in the section “Lifetime and Visibility of Variables” of Chapter 2.

In general, an identifier is visible only after it is defined. There are techniques in the language that allow declaring an identifier before its complete definition, but the general rule still applies if we consider both definitions and declarations.

44 - 01: Coding in Pascal

Given that it makes little sense to write an entire program in a single file, though, how does the rule above change when you use multiple units? In short, by referring to another unit with a `uses` statement, the identifiers in the interface section of that unit become visible to the new unit.

Reversing the perspective, if you declare an identifier (type, function, class, variable, and so on) in the interface portion of a unit, it becomes visible to any other module referring to that unit. If you declare an identifier in the implementation portion of a unit, instead, it can only be used in that unit (and it is generally referred to as a *local identifier*).

Using Units Like Namespaces

We have seen that the `uses` statement is the standard technique to access identifiers declared in the scope of another unit. At that point you can access the definitions of the unit. But it might happen that two units you refer to declare the same identifier; that is, you might have two classes or two routines with the same name.

In this case you can simply use the unit name to prefix the name of the type or routine defined in the unit. For example, you can refer to the `ComputeTotal` procedure defined in the given `Calc` unit as `Calc.ComputeTotal`. This isn't required often, as you are strongly advised against using the same identifier for two different elements of the same program, if you can avoid it.

However, if you look into the system or third party libraries, you'll find functions and classes that have the same name. A good example are the visual controls of different user interface frameworks. When you see a reference to `TForm` or `TControl`, it could mean different classes depending on the actual units you refer to.

If the same identifier is exposed by two units in your `uses` statement, the one in the last unit being used overrides the symbol, and will be the one that the compiler uses. In other words, the symbols defined in the last unit in the list wins. If you simply cannot avoid such a scenario, it is recommended to prefix the symbol with the unit name, to avoid having your code depend on the order in which the units are listed.

note Delphi developers do take advantage of the ability to have two classes with the same name, with a technique called *interposer classes* and explained later in this book.

The Program File

As we have seen, a Delphi application consists of two kinds of source code files: one or more units and one, and only one, program file (saved in a DPR file). The units can be considered secondary files, which are referenced by the main part of the application, the program. In theory, this is true. In practice, the program file is usually an automatically generated file with a limited role. It simply needs to start up the program, generally creating and running the main form, in case of a visual application. The code of the program file can be edited manually, but it is also modified automatically by using some of the Project Options of the IDE (like those related to the application object and the forms).

The structure of the program file is usually much simpler than the structure of the units. Here is the source code of a sample program file (with some optional standard units omitted) that is automatically created by the IDE for you:

```
program Project1;

uses
  FMX.Forms,
  Unit1 in 'Unit1.PAS' {Form1};

begin
  Application.Initialize;
  Application.CreateForm (TForm1, Form1);
  Application.Run;
end.
```

As you can see, there is simply a uses section and the main code of the application, enclosed by the begin and end keywords. The program's uses statement is particularly important, because it is used to manage the compilation and linking of the application.

tip

The list of units in the program file corresponds to the list of the units that are part of the project in the IDE Project Manager. When you add a unit to a project in the IDE, the unit is automatically added to the list in the program file source. The opposite happens if you remove it from the project. In any case, if you edit the source code of the program file, the list of units in the Project Manager is updated accordingly.

Compiler Directives

Another special element of the program structure (other than its actual code) are compiler directives, as mentioned earlier. These are special instructions for the compiler, written with the format:

```
| {$X+}
```

Some compiler directives have a single character, as above, with a plus or minus symbol indicating if the directive is activated or unactivated. Most of the directives also have a longer and readable version, and use `ON` and `OFF` to mark if they are active. Some directives have only the longer, descriptive format.

Compiler directives don't generate compiled code directly, but affect how the compiler generates code after the directive is encountered. In many cases, using a compiler directive is an alternative to changing one of the compiler settings in the IDE Project Options, although there are scenarios in which you want to apply a specific compiler setting only to a unit or to a fragment of code.

I'll cover specific compiler directives when relevant in the discussion of a language feature they can affect. In this section I only want to mention a couple of directives that relate to the program code flow: conditional defines and includes.

Conditional Defines

Conditional defines let you indicate to the compiler to include a portion of source code or ignore it. There are two flavors of conditional defines in Delphi. The traditional `$IFDEF` and `$IFNDEF` and the newer and more flexible `$IF`.

These conditional defines can depend on defined symbols or, for the `$IF` version, on constant values. The defined symbols can be predefined by the system (like compiler and the platform symbols), they can be defined in a specific project option, or they can be introduced in the code with another compiler directive, `$DEFINE`.

The traditional `$IFDEF` and `$IFNDEF` have this format:

```
{ $IFDEF TEST}
  // this is going to be compiled
{ $ENDIF}

{ $IFNDEF TEST}
  // this is not going to be compiled
{ $ENDIF}
```

You can also have two alternatives, using an `$ELSE` directive to separate them.

As mentioned, the newer `$IF` directive is more flexible, allowing the use of constant expressions for the condition, like comparison functions that can refer to any constant value in the code (for example checking if the compiler version is higher than a given value). The `$IF` directive is closed by an `$IFEND` directive:

```
{ $IF (ProgramVersion > 2.0) }
... // this code executes if condition is true
{ $ELSE }
... // this code execute if condition is false
{ $IFEND }
```

You can also use the `$ELSEIF` directive if you have multiple conditions.

Compiler Versions

Each version of the Delphi compiler has a specific define you can use to check if you are compiling against a specific version of the product. This might be required if you are using a feature introduced later but want to make sure the code still compiles for older versions.

If you need to have specific code for some of the recent versions of Delphi, you can base your `$IFDEF` statements on the following defines:

Delphi 2007	VER180
Delphi XE	VER220
Delphi XE2	VER230
Delphi XE4	VER250
Delphi XE5	VER260
Delphi XE6	VER270
Delphi XE7	VER280
Delphi XE8	VER290
Delphi 2007	VER180
Delphi XE	VER220
Delphi XE2	VER230

48 - 01: Coding in Pascal

Delphi XE4	VER250
Delphi XE5	VER260
Delphi XE6	VER270
Delphi XE7	VER280
Delphi XE8	VER290
Delphi 10 Seattle	VER300
Delphi 10.1 Berlin	VER310
Delphi 10.2 Tokyo	VER320
Delphi 10.3 Rio	VER330
Delphi 10.4 Sydney	VER340

The decimal digits of these version numbers indicate the actual compiler version (for example 26 in Delphi XE5). The numeric sequence is not specific to Delphi, but dates back to the first Turbo Pascal compiler published by Borland (see Appendix A for more information).

You can also use the internal versioning constant in `$IF` statements, with the advantage of being able to use a comparison operator (`>=`) rather than a match for a specific version. The versioning constant is called `CompilerVersion` and in Delphi XE5 it's assigned the floating-point value `26.0`.

So for example:

```
{ $IF CompilerVersion >= 26 }  
  // code to compile in Delphi XE5 or later  
{ $IFEND }
```

Similarly, you can use system defines for the different platforms you can compile for, in case you need some code to be platform-specific (generally an exception in Object Pascal, not common practice):

Windows (both 32 and 64 bit)	MSWINDOWS
macOS	MACOS
iOS	IOS

Android

ANDROID

Linux

LINUX

Below is a code snippet with the tests based on the platforms define above, part of the `HelloPlatform` project:

```
{ $IFDEF IOS}
  ShowMessage ('Running on iOS');
{ $ENDIF}

{ $IFDEF ANDROID}
  ShowMessage ('Running on Android');
{ $ENDIF}
```

Include Files

The other directive I want to cover here is the `$INCLUDE` directive, already mentioned when discussing the `uses` statement. This directive lets you refer to and include a fragment of source code in a given position of a source code file. At times this is used to be able to include the same fragment in different units, in cases where the code fragment defines compiler directives and other elements used directly by the compiler.

When you use a unit, it is compiled only once. When you include a file, that code is compiled within each of the units it is added to (which is why you should generally avoid having any new identifier declared in an include file, if the same include file is embedded in different units of the same project. So how do you use an include file? A good example is a set of compiler directives you want to enable in most of your units, or some extra special defines.

Large libraries often use include files for that purpose, an example would be the FireDAC library, a database library which is now part of the system libraries. Another example, showcased by the system RTL units, is the use of individual includes for each platform, with an `$IFDEF` used for conditionally including only one of them.

50 - 01: Coding in Pascal

02: variables and data types

Object Pascal is what is known as a *strongly-typed* language. Variables in Object Pascal are declared to be of a *data type* (or *user defined data type*). The type of a variable determines the values a variable can hold, and the operations that can be performed on it. This allows the compiler both to identify errors in your code and generate faster programs for you.

This is why the concept of type is stronger in Pascal than in languages like C or C++. Later languages based on the same syntax but that break compatibility with C, like C# and Java, diverted from C and embraced Pascal's strong notion of data type. In C, for example, arithmetic data types are almost interchangeable. By contrast the original versions of BASIC, had no similar concept, and in many of today's scripting languages (JavaScript being an obvious example) the notion of data type is very different.

note In fact, there are some tricks to bypass type safety, like using variant record types. The use of these tricks is strongly discouraged and they aren't much used today.

52 - 02: Variables and Data Types

As I mentioned, all of the dynamic languages, from JavaScript onwards, don't have the same notion of data type, or (at least) have a very loose idea of types. In some of these languages the type is inferred by the value you assign to a variable, and the variable type can change over time. What is important to point out is that data types are a key element for enforcing correctness of a large application at compile-time, rather than relying on run-time checks. Data types require more order and structure, and some planning of the code you are going to write – which clearly has advantages and disadvantages.

note Needless to say I prefer strongly typed languages, but in any case my goal in this book is to explain how the language works, more than to advocate why I think it is such a great programming language. Though I'm sure you'll get that impression while you read the text.

Variables and Assignments

Like other strongly-typed languages, Object Pascal requires all variables to be declared before they are used. Every time you declare a variable, you must specify a data type. Here are some variable declarations:

```
var
  Value: Integer;
  IsCorrect: Boolean;
  A, B: Char;
```

The `var` keyword can be used in several places in a program, such as at the beginning of a function or procedure, to declare variables local to that portion of the code, or inside a unit to declare global variables.

note As we'll see in a bit, recent versions of Delphi add the ability to declare a variable “inline”, that is mixed into programming statements. This is a significant departure from classic Pascal.

After the `var` keyword comes a list of variable names, followed by a colon and the name of the data type. You can write more than one variable name on a single line, as `A` and `B` in the last statement of the previous code snippet (a coding style that is less common today, compared to splitting the declaration on two separate lines: using separate lines helps with readability, versions comparison and merging).

Once you have defined a variable of a given type, you can only perform the operations supported by its data type on it. For example, you can use the Boolean value in a test and the integer value in a numerical expression. You cannot mix Boolean val-

ues and Integers without calling a specific conversion function, for example, or mix any incompatible data type (even if the internal representation might be physically compatible, as it is the case for Boolean values and Integers).

The simplest assignment is that of an actual, specific value. For example, you might want the `value` variable to hold the numeric value 10. But how do you express literal values? While this concept might be obvious, it is worth looking into it with some detail.

Literal Values

A literal value is a value you type directly in the program source code. If you need a number with the value of twenty, you can simply write:

```
| 20
```

There is also an alternative representation of the same numeric value, based on the hexadecimal value, like:

```
| $14
```

These will be literal values for an integer number (or one of the various integer number types that are available). If you want the same numeric value, but for a floating point literal value, you generally add a zero decimal after it:

```
| 2.0
```

Literal values are not limited to numbers. You can also have characters and strings. Both use single quotes (were many other programming languages will use double quotes for both, or single quotes for characters and double quotes for strings):

```
| // literal characters
| 'K'
| #55
|
| // literal string
| 'Marco'
```

As you can see above, you can also indicate characters by their corresponding numeric value (originally the ASCII number, now the Unicode code point value), prefixing the number with the `#` symbol, as in `#32` (for a space). This is useful mostly for control characters without a textual representation in the source code, like a backspace or a tab.

In case you need to have a quote within a string, you'll have to double it. So if I want to have my first and last name (spelled with a final quote rather than an accent) I can write:

```
| 'My name is Marco Cantu'''
```

54 - 02: Variables and Data Types

The two quotes stand for a quote within the string, while the third consecutive quote marks the end of the string. Also note that a string literal must be written on a single line, but you can concatenate multiple string literals using the + sign. If you want to have the new line or line break within the string, don't write it on two lines, but concatenate the two elements with the `sLineBreak` system constant (which is platform specific), as in:

```
| 'Marco' + sLineBreak + 'Cantu'''
```

Assignment Statements

Assignments in Object Pascal use the colon-equal operator (`:=`), an odd notation for programmers who are used to other languages. The `=` operator, which is used for assignments in many other languages, in Object Pascal is used to test for equality.

history The `:=` operator comes from a Pascal predecessor, Algol, a language few of today's developers have heard of (let even used). Most of today's languages avoid the `:=` notation and favor the `=` assignment notation.

By using different symbols for an assignment and an equality test, the Pascal compiler (like the C compiler) can translate source code faster, because it doesn't need to examine the context in which the operator is used to determine its meaning. The use of different operators also makes the code easier for people to read. Truly Pascal picked two different operators than C (and syntactic derivatives like Java, C#, JavaScript), which uses `=` for assignment and `==` for equality testing.

note For the sake of completeness I should mention JavaScript has also a `===` operator (performing a strict type and value equality test), but that's something that even most JavaScript programmers get confused about.

The two elements of an assignment are often called *lvalue* and *rvalue*, for left value (the variable or memory location you are assigning to) and right value, the value of the expressions being assigned. While the *rvalue* can be an expression, the *lvalue* must refer (directly or indirectly) to a memory location you can modify. There are some data types that have specific assignment behaviors which I'll cover in due time.

The other rule is that the type of the *lvalue* and of the *rvalue* must match, or there must be an automatic conversion between the two, as explained in the next section.

Assignments and Conversion

Using simple assignments, we can write the following code (that you can find among many other snippets in this section in the `variablesTest` project):

```
Value := 10;
Value := Value + 10;
IsCorrect := True;
```

Given the previous variable declarations, these three assignments are correct. The next statement, instead, is not correct, as the two variables have different data types:

```
Value := IsCorrect; // error
```

As you type this code, the Delphi editor immediately shows a *red squiggle* indicating the error, with a proper description. If you try to compile it, the compiler issues the same error with a description like:

```
[dcc32 Error]: E2010 Incompatible types: 'Integer' and 'Boolean'
```

The compiler informs you that there is something wrong in your code, namely two incompatible data types. Of course, it is often possible to convert the value of a variable from one type to another type. In some cases, the conversion is automatic, for example if you assign an integer value to a floating point variable (but not the opposite, of course). Usually you need to call a specific system function that changes the internal representation of the data.

Initializing Global Variables

For global variables, you can assign an initial value as you declare the variable, using the constant assignment notation covered below (=) instead of the assignment operator (:=). For example, you can write:

```
var
  Value: Integer = 10;
  Correct: Boolean = True;
```

This initialization technique works only for global variables, which are initialized to their default values anyway (like to zero for a number).

Initializing Local Variables

Variables declared at the beginning of a procedure or function, instead, are *not* initialized to a default value and don't have an assignment syntax. For those variables, it is often worth adding explicit initialization code at the beginning of the code:

```
var
  value: Integer;
begin
  value := 0; // initialize
```

Again, if you don't initialize a local variable but use it as is, the variable will have a totally random value (depending on the bytes that were present at that memory location). In several scenarios, the compiler will warn you of the potential error, but not always.

In other words, if you write:

```
var
  value: Integer;
begin
  ShowMessage (value.ToString); // value is undefined
```

The output will be a totally random value, whatever bytes happened to be at the memory location of the `value` variable considered as an `Integer`.

Inline Variables

Recent versions of Delphi (starting with 10.3 Rio) have an additional concept, which changes the variable declaration approach used since the early Pascal and Turbo Pascal compilers: inline variable declarations.

The new inline variable declaration syntax allows you to declare the variable directly in a code block (allowing also multiple symbols as for traditional variable declarations):

```
procedure Test;
begin
  var I, J: Integer;
  I := 22;
  J := I + 20;
  ShowMessage (J.ToString);
end;
```

While this might seem a limited difference, there are several side effects of this change related with initialization, type inference, and lifetime of the variable. I'm going to cover those in coming sections of this chapter.

Initializing Inline Variables

The first significant change compared to the old declaration model, is that the inline declaration and initialization of a variable can be done in a single statement. This makes things more readable compared to initializing several variables at the beginning of a function, as indicated in the previous section:

```
procedure Test;
begin
  var I: Integer := 22;
  ShowMessage (I.ToString);
end;
```

The main advantage here is that if the value of a variable is becoming available only later in the code block, rather than setting an initial value (like 0 or nil) and later assigning the actual value, you can delay the variable declaration to the point where you can calculate a proper initial value, like κ below:

```
procedure Test1;
begin
  var I: Integer := 22;
  var J: Integer := 22 + I;
  var K: Integer := I + J;
  ShowMessage (K.ToString);
end;
```

In other words, while in the past all local variables were visible in the entire code block, now an inline variable is visible only from the position of its declaration and up to the end of the code block. In other words, a variable like κ cannot be used in the first 2 lines of the code block, before it is assigned a proper value.

Type Inference for Inline Variables

Another significant benefit of inline variables is that the compiler now can, in several circumstances, infer the type of an inline variable by looking at the type of the expression or value assigned to it. This is a very simple example:

```
procedure Test;
begin
  var I := 22;
  ShowMessage (I.ToString);
end;
```

The type of the *rvalue* expression (that is, what comes after the :=) is analyzed to determine the type of the variable. By assigning a string constant, the variable would be a string type, but the same analysis happens if you assign the result of a function or a complex expression.

58 - 02: Variables and Data Types

Notice that the variable is still strongly type, as the compiler determines the data type and assign it at compile time and it cannot be modified by assigning a new value. Type inference is only a convenience to type less code (something relevant for complex data type, like complex generic types) but doesn't change the static and strongly typed nature of the language or cause any slow down at run time.

note When inferring the type, some of the data types are “expanded” to a larger type, as in the case above where the numeric value 22 (a `ShortInt`) is expanded to `Integer`. As a general rule, if the right hand expression type is an integer type and smaller than 32 bits, the variable will be declared as a 32-bit `Integer`. You can use an explicit type if you want a specific, smaller, numeric type.

Constants

Object Pascal also allows the declaration of constants. This allows you to give meaningful names to values that do not change during program execution (and possibly reducing the size by not duplicating constant values in your compiled code).

To declare a constant you don't need to specify a data type, but only assign an initial value. The compiler will look at the value and automatically infer the proper data type. Here are some sample declarations (also from the `variableTest` application project):

```
const
  Thousand = 1000;
  Pi = 3.14;
  AuthorName = 'Marco Cantu';
```

The compiler determines the constant data type based on its value. In the example above, the `Thousand` constant is assumed to be of type `SmallInt`, the smallest integral type that can hold it. If you want to tell the compiler to use a specific type you can simply add the type name to the declaration, as in:

```
const
  Thousand: Integer = 1000;
```

warning Assumptions in a program are generally bad, and the compiler might change over time not match the developer assumption any more. If you have a way to express the code more clearly and with no assumptions, do it!

When you declare a constant, the compiler can choose whether to assign a memory location to the constant and save its value there, or to duplicate the actual value each time the constant is used. This second approach makes sense particularly for simple constants.

Once you have declared a constant you can use it almost like any other variable, but you cannot assign a new value to it. If you try, you'll get a compiler error.

note Oddly enough, Object Pascal does allow you to change the value of a typed constant at run-time, as if it was a variable but only if you enable the `$J` compiler directive, or use the corresponding *Assignable typed constants* compiler option. This optional behavior is included for backward compatibility of code which was written with an old compiler. This is clearly not a suggested coding style, and I've covered it in this note most as a historical anecdote about such programming techniques.

Inline Constants

As we have seen earlier for variables, you can now also inline a constant value declaration. This can be applied to typed constants or untyped constants, in which case the type is inferred (a feature that has been available for constants for a long time). A simple example is below:

```
begin
  // some code
  const M: Integer = (L + H) div 2; // identifier with type specifier
  // some code
  const M = (L + H) div 2; // identifier without type specifier
```

Notice that while a regular constant declaration allows you only to assign a constant value, for an inline constant declaration you can use any expression.

Resource String Constants

Although this is a slightly more advanced topic, when you define a string constant, instead of writing a standard constant declaration you can use a specific directive, `resourcestring`, that indicates to the compiler and linker to treat the string like a Windows resource (or an equivalent data structure on non-Windows platforms Object Pascal supports):

```
const
  AuthorName = 'Marco';

resourcestring
  StrAuthorName = 'Marco';

begin
  ShowMessage (StrAuthorname);
```

In both cases you are defining a constant; that is, a value you don't change during program execution. The difference is only in the internal implementation. A string constant defined with the `resourcestring` directive is stored in the resources of the program, in a string table.

60 - 02: Variables and Data Types

In short, the advantages of using resources are more efficient memory handling performed by Windows, a corresponding Delphi implementation for other platforms, and a better way of localizing a program (translating the strings to a different language) without having to modify its source code. As a rule of thumb, you should use `resourcestring` for any text that is shown to users and might need translating, and internal constants for every other internal program string, like a fixed configuration file name.

tip The IDE editor has an automatic *refactoring* you can use to replace a string constant in your code with a corresponding `resourcestring` declaration. Place the edit cursor within a string literal and press Ctrl+Shift+L to activate this refactoring.

Lifetime and Visibility of Variables

Depending on how you define a variable, it will use different memory locations and remain available for a different amount of time (something generally called the variable *lifetime*) and will be available in different portions of your code (a feature referred to by the term *visibility*).

Now, we cannot have a complete description of all of the options this early in the book, but we can certainly consider the most relevant cases:

- **Global variables:** If you declare a variable (or any other identifier) in the interface portion of the unit, its scope extends to any other unit that uses the one declaring it. The memory for this variable is allocated as soon as the program starts and exists until it terminates. You can assign a default value to it or use the initialization section of the unit in case the initial value is computed in a more complex way.
- **Global hidden variables:** If you declare a variable in the implementation portion of a unit, you cannot use it outside that unit, but you can use it in any block of code and procedure defined within the unit, from the position of the declaration onwards. Such a variable uses global memory and has the same lifetime as the first group; the only difference is in its visibility. The initialization is the same as that of global variable.
- **Local variables:** If you declare a variable within the block defining a function, procedure, or method, you cannot use this variable outside that block of code. The scope of the identifier spans the whole function or method, including nested routines (unless an identifier with the same name in the nested routine hides the outer definition). The memory for this variable is allocated on the stack when the program executes the routine defining it. As

soon as the routine terminates, the memory on the stack is automatically released.

- **Local inline variables:** If you declared an inline variable within the block defining a function, procedure, or method, the additional restriction compared to a traditional local variable is that the visibility starts from the line the variable is declared and spans until the end of the function or method.
- **Block-scoped inline variables:** If you declare an inline variable within a code block (that is, a nested `begin-end` statement) the visibility of the variable will be limited to that nested block. This matches what happens in most other programming languages, but was introduced in Object Pascal only with the recent inline variable declaration syntax. Notice that an inline variable declared in a block cannot use the same identifier as a variable declared in the parent code block.

note For block-scoped inline variables, not only the visibility but also the variable lifetime is limited to the block. A managed data type (like an interface, string or managed record) will get disposed at the end of the sub-block, rather than at the end of the procedure or method. This is also true for temporary variables created to hold expression results.

Any declarations in the interface portion of a unit are accessible from any part of the program that includes the unit in its `uses` clause. Variables of form classes are declared in the same way, so that you can refer to a form (and its public fields, methods, properties, and components) from the code of any other form. Of course, it's poor programming practice to declare everything as global. Besides the obvious memory consumption problems, using global variables makes a program harder to maintain and update. In short, you should use the smallest possible number of global variables.

Data Types

In Pascal there are several predefined data types, which can be divided into three groups: *ordinal types*, *real types*, and *strings*. We'll discuss ordinal and real types in the following sections, while strings will be specifically covered in Chapter 6.

Delphi also includes a *non-typed* data type, called *variant*, and other “flexible” types, such as `TValue` (part of the enhanced RTTI support). Some of these more advanced data types will be discussed later in Chapter 5.

Ordinal and Numeric Types

Ordinal types are based on the concept of order or sequence. Not only can you compare two values to see which is higher, but you can also ask for the next or previous values of any value and compute the lowest and highest possible values the data type can represent.

The three most important predefined ordinal types are `Integer`, `Boolean`, and `Char` (character). However, there are other related types that have the same meaning but a different internal representation and support a different range of values.

The following table lists the ordinal data types used for representing numbers:

Size	Signed	Unsigned
8 bits	<code>ShortInt</code> : -128 to 127	<code>Byte</code> : 0 to 255
16 bits	<code>SmallInt</code> : -32768 to 32767 (-32K to 32K)	<code>Word</code> : 0 to 65,535 (0 to 64K)
32 bits	<code>Integer</code> : -2,147,483,648 to 2,147,483,647 (-2GB to +2GB)	<code>Cardinal</code> : 0 to 4,294,967,295 (0 to 4 GB)
64 bits	<code>Int64</code> : -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	<code>UInt64</code> : 0 to 18,446,744,073,709,551,615 (if you can read it!)

As you can see, these types correspond to different representations of numbers, depending on the number of bits used to express the value, and the presence or absence of a sign bit. Signed values can be positive or negative, but have a smaller range of values (half of the corresponding unsigned value), because one less bit is available for storing the value itself.

The `Int64` type represents integer numbers with up to 18 digits. This type is fully supported by some of the ordinal type routines (such as `High` and `Low`), numeric routines (such as `Inc` and `Dec`), and string-conversion routines (such as `IntToStr`) of the run time library.

Aliased Integer Types

If you have a hard time remembering the difference between a `ShortInt` and a `SmallInt` (including which one is effectively smaller), rather than the actual type you can use one of the predefined aliases declared in the `System` unit:

```
type
  Int8   = ShortInt;
  Int16  = SmallInt;
  Int32  = Integer;
  UInt8  = Byte;
  UInt16 = Word;
  UInt32 = Cardinal;
```

Again, these types don't add anything new, but are probably easier to use, as it is simple to remember the actual implementation of an `Int16` rather than that of a `SmallInt`. These type aliases are also easier to use for developers coming from C and other languages that use similar type names.

Integer Type, 64 Bit, NativeInt, and LargeInt

In 64-bit versions of Object Pascal you may be surprised to learn that the `Integer` type is still 32 bit. It is so because this is the most efficient type for numeric processing at the CPU level.

It is the `Pointer` type (more about pointers later on) and other related reference types that are 64 bit. If you need a numeric type that adapts to the pointer size and the native CPU platform, you can use the two special `NativeInt` and `NativeUInt` aliased types. These are the same size of a pointer on the specific platform (that is, 32 bit on 32-bit platforms and 64 bit on 64-bit platforms).

A slightly different scenario happens for the `LargeInt` type, which is often used to map to native platform API functions. This is 32 bit on 32-bit platforms and on Windows 32 bit, while it is 64 bit on 64-bit ARM platform. Better stay away from it unless you need it specifically for native code in a way it adapts to the underlying operating system.

Integer Type Helper

While the Integer types are treated separately from objects in the Object Pascal language, it is possible to operate on variables (and constant values) of these types with operations that you apply using “dot notation”. This is the notation generally used to apply methods to objects.

64 - 02: Variables and Data Types

note Technically these operations on native data types are defined using “intrinsic record helpers”. Class and record helpers are covered in Chapter 12. In short, you can customize the operations applicable to core data types. Expert developers will notice that type operations are defined as class static methods in the matching intrinsic record helper.

You can see a couple of examples in the following code extracted from the `IntegerTest` demo:

```
var
  N: Integer;
begin
  N := 10;
  Show (N.ToString);

  // display a constant
  Show (33.ToString);

  // type operation, show the bytes required to store the type
  Show (Integer.Size.ToString);
```

note The `Show` function used in this code snippet is a simple procedure used to display some string output in a memo control, to avoid having to close multiple `ShowMessage` dialogs. A side advantage is this approach makes easier to copy the output and paste in the text (as I've done below). You'll see this approach used through most of the demos of this book.

The output of the program is the following

```
10
33
4
```

Given these operations are very important (more than others that are part of the run time library) it is worth listing them here:

<code>ToString</code>	Convert to the number to a string, using a decimal format
<code>ToBoolean</code>	Conversion to Boolean type
<code>ToHexString</code>	Convert to a string, using a hexadecimal format
<code>ToSingle</code>	Conversion to single floating point data type
<code>ToDouble</code>	Conversion to double floating point data type
<code>ToExtended</code>	Conversion to extended floating point data type

The first and third operations convert to the number to a string, using a decimal or hexadecimal operation. The second is a conversion to Boolean, while the last three are conversions to floating point types described later.

There are other operations you can apply to the `Integer` type (and most other numerical types), such as:

<code>Size</code>	The number of bytes required to store a variable of this type
-------------------	---------------------------------------------------------------

<code>Parse</code>	Convert a string to the numeric value it represents, raising an exception if the string doesn't represent a number
<code>TryParse</code>	Try to convert the string to a number

Standard Ordinal Type Routines

Beside the operations defined by Integer type helpers and listed above, there are several standard and “classic” functions you can apply to any ordinal type (not just the numeric ones). A classic example is asking for information about the type itself, using the functions `SizeOf`, `High`, and `Low`. The result of the `SizeOf` system function (that you can apply to any data type of the language) is an integer indicating the number of bytes required to represent values of the given type (just like the `Size` helper function shown above)

The system routines that work on ordinal types are shown in the following table:

<code>Dec</code>	Decrements the variable passed as parameter, by one or by the value of the optional second parameter
<code>Inc</code>	Increments the variable passed as parameter, by one or by the specified value
<code>Odd</code>	Returns True if the argument is an odd number. For testing for even numbers, you should use a not expression (<code>not Odd</code>)
<code>Pred</code>	Returns the value before the argument in the order determined by the data type, the predecessor
<code>Succ</code>	Returns the value after the argument, the successor
<code>Ord</code>	Returns a number indicating the order of the argument within the set of values of the data type (used for non-numerical ordinal types)
<code>Low</code>	Returns the lowest value in the range of the ordinal type passed as parameter
<code>High</code>	Returns the highest value in the range of the ordinal data type

note C and C++ programmers should notice that the two versions of the `Inc` procedure, with one or two parameters, correspond to the `++` and `+=` operators (the same holds for the `Dec` procedure which corresponds to the `--` and `-=` operators). The Object Pascal compiler optimizes these increment and decrement operations, similarly to the way C and C++ compilers do. However, Delphi offers only pre-increment and pre-decrement versions, not the post-increment and post-decrement, and the operation doesn't return a value.

Notice that some of these routines are automatically evaluated by the compiler and replaced with their value. For example, if you call `High(x)` where `x` is defined as an Integer, the compiler replaces the expression with the highest possible value of the Integer data type.

66 - 02: Variables and Data Types

In the `IntegersTest` application project I've added an event with a few of these ordinal type functions:

```
var
  N UInt16;
begin
  N := Low (UInt16);
  Inc (N);
  Show (N.ToString);
  Inc (N, 10);
  Show (N.ToString);
  if odd (N) then
    Show (N.ToString + ' is odd');
```

This is the output you should see:

```
1
11
11 is odd
```

You can change the data type from `UInt16` to `Integer` or other ordinal types to see how the output changes.

Out-Of-Range Operations

A variable like `n` above has only a limited range of valid values. If the value you assign to it is negative or too big, this results in an error. There are actually three different types of errors you can encounter with out-of-range operations.

The first type of error is a compiler error, which happens if you assign a constant value (or a constant expression) that is out of range. For example, if you add to the code above:

```
N := 100 + High (N);
```

the compiler will issue the error:

```
[dcc32 Error] E1012 Constant expression violates subrange bounds
```

The second scenario takes place when the compiler cannot anticipate the error condition, because it depends on the program flow. Suppose we write (in the same piece of code):

```
Inc (N, High (N));
Show (N.ToString);
```

The compiler won't trigger an error because there is a function call, and the compiler doesn't know its effect in advance (and the error would also depend on the initial value of `n`). In this case there are two possibilities. By default, if you compile and run this application, you'll end up with a completely illogical value in the variable (in this case the operation will result in subtracting 1!). This is the worst possible scenario, as you get no error, but your program is not correct.

What you can do (and it is highly suggested to do) is to turn on a compiler option called “Overflow checking” (`{Q+}` or `{OVERFLOWCHECKS ON}`), which will guard against a similar *overflow* operation and raise an error, in this specific case “Integer overflow”.

Boolean

Logical `True` and `False` values are represented using the Boolean type. This is also the type of the condition in conditional statements, as we’ll see in the next chapter. The Boolean type can only have one of the two possible values `True` and `False`.

warning For compatibility with Microsoft’s COM and OLE automation, the data types `ByteBool`, `WordBool`, and `LongBool` represent the value `True` with `-1`, while the value `False` is still `0`. Again, you should generally ignore these types and avoid all low-level Boolean manipulation and numeric mapping unless absolutely necessary.

Unlike in the C language and some of its derived languages, Boolean is an enumerated type in Object Pascal, there is no direct conversion to the numeric value representing a Boolean variable, and you should not abuse direct type casts by trying to convert a Boolean to a numeric value. It is true, however, that Boolean type helpers include the functions `ToInteger` and `ToString`. I cover enumerated types later in this chapter.

Notice that using `ToString` returns the string with the numeric value of the Boolean variable. As an alternative you can use the `BoolToStr` global function, setting the second parameter to `True`, to indicate the use of Boolean strings (`'True'` and `'False'`) for the output. (See the section “Char Type Operations” below for an example.)

Characters

Character variable are defined using the Char type. Unlike older versions, the language today uses the Char type to represent double-byte Unicode characters (also represented by the `wideChar` type).

note The Delphi compiler still offers the distinction between `AnsiChar` for one byte ANSI characters and `wideChar` for Unicode ones, with the `Char` type defined as an alias of the latter. The recommendation is to focus on `wideChar`, and use the `Byte` data type for single byte elements. However, it is true that starting with Delphi 10.4 the `AnsiChar` type has been made available on all compilers and platform for better compatibility with existing code.

68 - 02: Variables and Data Types

For an introduction to characters in Unicode, including the definition of a code point and that of surrogate pairs (among other advanced topics) you can read Chapter 6. In this section I'll just focus on the core concepts of the Char type.

As I mentioned earlier while covering literal values, constant characters can be represented with their symbolic notation, as in 'k', or with a numeric notation, as in #78. The latter can also be expressed using the Chr system function, as in Chr (78). The opposite conversion can be done with the Ord function. It is generally better to use the symbolic notation when indicating letters, digits, or symbols.

When referring to special characters, like the control characters below #32, you'll generally use the numeric notation. The following list includes some of the most commonly used special characters:

#8	backspace
#9	tab
#10	new line
#13	carriage return
#27	escape

Char Type Operations

As other ordinal types, the Char type has several predefined operations you can apply to variables of this type using the dot notation. These operations are defined with an intrinsic record helper, again.

However, the usage scenario is quite different. First, to use this feature you need to *enable* it by referring to the Character unit in a uses statement. Second, rather than a few conversion functions, the helper for the Char type includes a couple of dozen Unicode-specific operations, including tests like IsLetter, IsNumber, and IsPunctuation, and conversions like ToUpper and ToLower. Here is an example taken from the CharTest application project:

```
uses
  Character;
...
var
  Ch: Char;
begin
  Ch := 'a';
  Show (BoolToStr(Ch.IsLetter, True));
  Show (Ch.ToUpper);
```

The output of this code is:

```
True
A
```

note The `Toupper` operation of the `Char` type helper is fully Unicode enabled. This means that if you pass an accented letter like `ù` the result will be `Û`. Some of the traditional RTL functions are not so smart and work only for plain ASCII characters. They haven't been modified so far because the corresponding Unicode operations are significantly slower.

Char as an Ordinal Type

The `Char` data type is quite large, but it is still an ordinal type, so you can use `Inc` and `Dec` functions on it (to get to the next or previous character or move ahead by a given number of elements, as we have seen in the section “Standard Ordinal Types Routines”) and write `for` loops with a `Char` counter (more on `for` loops in the next chapter).

Here is a simple fragment used to display a few characters, obtained by increasing the value from a starting point:

```
var
  Ch: Char;
  Str1: string;
begin
  Ch := 'a';
  Show (Ch);
  Inc (Ch, 100);
  Show (Ch);

  Str1 := '';
  for Ch := #32 to #1024 do
    Str1 := Str1 + Ch;
  Show (Str1)
```

The `for` loop of the `CharsTest` application project adds a lot of text to the string, making the output quite long. It starts with the following lines of text:

```
a
À
!"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]^_`abc
defghijklmnopqrstuvwxyz{|}~
// few more lines omitted...
```

Converting with Chr

We have seen that there is an `Ord` function that returns the numeric value (more specifically a Unicode code point) of a character. There is also an opposite function you can use to get the character corresponding to a code point, that is the `Chr` special function.

32-bit Characters

Although the default Char type is now mapped to WideChar, it is worth noticing that Delphi also defines a 4-byte character type, UCS4Char, defined in the System unit as:

```
type  
  UCS4Char = type LongWord;
```

This type definition and the corresponding one for UCS4String (defined as an array of UCS4Char) are little used, but they are part of the language runtime and used in some of the functions of the Character unit.

Floating Point Types

While integer numbers of various kinds can be represented with an ordinal set of values, floating point numbers are not ordinal (they have the concept of order, but not the concept of a sequence of elements) and represented by some approximate value, with some error in their representation.

Floating-point numbers come in various formats, depending on the number of bytes used to represent them and the quality of the approximation. Here is a list of floating-point data types in Object Pascal:

Single	The smallest storage size is given by Single numbers, which are implemented with a 4-byte value. The name indicates a single precision floating point value and the same type is indicated with the name float in other languages.
Double	These are floating-point numbers implemented with 8 bytes. The name indicates a double precision floating point value and is shared by many languages. The Double precision is the most commonly used floating point data type and is also an alias of an older Pascal type called Real.
Extended	These are numbers implemented with 10 bytes in the original Delphi Win32 compiler, but this type is not available on all platforms (on some, like Win64, it reverts back to Double, while on macOS it is 16 bytes). Other programming languages call this data type long double.

These are all floating-point data types with different precision, which correspond to the IEEE standard floating-point representations, and are directly supported by the CPU (or, to be precise, by the FPU, the floating point unit), for maximum speed.

There are also two peculiar non-ordinal numeric data types you can use to represent numbers with a precise, not an approximate representation:

<code>Comp</code>	Describes very big integers using 8 bytes or 64 bits (which can hold numbers with 18 decimal digits). The idea is to represent large numbers with no loss of precision, unlike the corresponding floating point values.
<code>Currency</code>	Indicates a fixed-point decimal value with four decimal digits, and the same 64-bit representation as the <code>Comp</code> type. As the name implies, the <code>Currency</code> data type has been added to handle very precise monetary values, with four decimal places (again with no loss of precision in calculations).

All of these non-ordinal data types don't have the concepts of the `High`, `Low`, or `Ord` function. Real types represent (in theory) an infinite set of numbers; ordinal types represent a fixed set of values.

Why Floating Point Values are Different

Let me explain further. When you have the integer 23 you can determine which is the following value. Integers are finite (they have a determined range and they have an order). Floating point numbers are infinite even within a small range, and have no sequence: in fact, how many values are there between 23 and 24? And which number follows 23.46? Is it 23.47, 23.461, or 23.4601? That's really impossible to know!

For this reason, whilst it makes sense to ask for the ordinal position of the character 'w' in the range of the `Char` data type, it makes no sense at all to ask the same question about 7143.1562 in the range of a floating-point data type. Although you can indeed know whether one real number has a higher value than another, it makes no sense to ask how many real numbers exist before a given number (this is the meaning of the `Ord` function).

Another key concept behind floating point values is that their implementation cannot represent all numbers precisely. It is often the case that the result of a calculation you'd expect to be a specific number (at times an integer one), could in fact be an approximate value of it. Consider this code, taken from the `FloatTest` application project:

```
var
  s1: Single;
begin
  s1 := 0.5 * 0.2;
  Show (s1.ToString);
```


72 - 02: Variables and Data Types

You would expect the result to be 0.1, while in fact you'd get something like 0.100000001490116. This is close to the expected value, but not exactly it. Needless to say, if you round the result, you'll get the expected value. If you use a `Double` variable, instead, the output will be 0.1, as the `FloatTest` application project shows.

note Now I don't have time for an in-depth discussion of floating point math on computers, so I'm cutting this discussion rather short, but if you are interested in this topic from the Object Pascal language perspective, I can recommend you an excellent article from the late Rudy Velthuis at <http://rvelthuis.de/articles/articles-floats.html>.

Floating Helpers and the Math Unit

As you can see from the code snippet above, the floating point data types also have record helpers allowing you to apply operations directly to the variables, as if they were objects. In fact, the list of operations for floating point numbers is actually quite long.

This is the list of operations on instances for the `Single` type (with some operations obvious from their names, others a bit more cryptic you can look up in the documentation):

Exponent	Fraction	Mantissa
Sign	Exp	Frac
SpecialType	Buildup	ToString
IsNan	IsInfinity	IsNegativeInfinity
IsPositiveInfinity	Bytes	Words

The run-time library also has a `Math` unit that defines advanced mathematical routines, covering trigonometric functions (such as the `ArcCosh` function), finance (such as the `InterestPayment` function), and statistics (such as the `MeanAndStdDev` procedure). There are a number of these routines, some of which sound quite strange to me, such as the `MomentSkewKurtosis` function (I'll let you find out what this is).

The `System.Math` unit is very rich in capabilities, but you'll also find many external collections of mathematical functions for Object Pascal.

Simple User-Defined Data Types

Along with the notion of type, one of the great ideas introduced by Niklaus Wirth in the original Pascal language was the ability to define new data types in a program (something we take for granted today, but wasn't obvious at the time). You can

define your own data types by means of *type definitions*, such as subrange types, array types, record types, enumerated types, pointer types, and set types. The most important user-defined data type is the class, which is part of the object-oriented capabilities of the language, covered in the second part of this book.

If you think that type constructors are common in many programming languages, you are right, but Pascal was the first language to introduce the idea in a formal and very precise way. Object Pascal still has some rather unique capabilities, like the definition of subrange, enumerations, and sets, covered in the following sections. More complex data type constructors (like arrays and records) are covered in Chapter 5.

Named vs. Unnamed Types

User-defined data types can be given a name for later use or applied to a variable directly. The convention in Object Pascal is to use a letter τ prefix to denote any data type, including classes but not limited to them. I strongly suggest you to stick to this rule, even if might not feel natural at first if you are coming from a Java or C# background.

When you give a name to a type, you must do so in a “type” section of your program (you can add as many types as you want in each unit). Below is a simple example of a few type declarations:

```
type
  // subrange definition
  TUppercase = 'A'..'Z';

  // enumerated type definition
  TMyColor = (Red, Yellow, Green, Cyan, Blue, Violet);

  // set definition
  TColorPalette = set of TMyColor;
```

With these types, you can now define some variables:

```
var
  UpSet: TUpperLetters;
  Color1: TMyColor;
```

In the scenario above I'm using a *named* type. As an alternative, the type definition can be used directly to define a variable without an explicit type name, as in the following code:

```
var
  Palette: set of TMyColor;
```

74 - 02: Variables and Data Types

In general, you should avoid using *unnamed* types as in the code above, because you cannot pass them as parameters to routines or declare other variables of the same type. Given the language ultimately resorts to *type name equivalence* rather than structural type equivalence, having a single definition for each type is indeed important. Also remember that type definitions in the `interface` portion of a unit can be seen in the code of any other units by means of a `uses` statement.

What do the type definitions above mean? I'll provide some descriptions for those who are not familiar with traditional Pascal type constructs. I'll also try to underline the differences from the same constructs in other programming languages, so you might be interested in reading the following sections in any case.

Type Aliases

As we've seen, the Delphi language uses the type name (not its actual definition) when checking for type compatibility. Two identically defined types with different names, are two different types.

This is partially true also when you define a type alias, that is a new type name based on an existing type. What is confusing is that there are two variations of the same syntax which produce slightly different effects. Look at this code in the `TypeAlias` example:

```
type
  TNewInt = Integer;
  TNewInt2 = type Integer;
```

Both new types remain assignment compatible with the `Integer` type (via automatic conversion), however the `TNewInt2` type won't be an exact match and, for example, it cannot be passed as reference parameter to a function expecting the aliased type:

```
procedure Test (var N: Integer);
begin

end;

procedure TForm40.Button1Click(Sender: TObject);
var
  I: Integer;
  NI: TNewInt;
  NI2: TNewInt2;
begin
  I := 10;
  NI := I; // works
  NI2 := I; // works

  Test(I);
```

```
Test (NI);
Test (NI2); // error
```

The last line produces the error:

```
E2033 Types of actual and formal var parameters must be identical
```

Something similar happens with type helpers, as the Integer type helper can be used for `newint` but not for `TNewInt2`. This is specifically covered in a later section when discussing record helpers.

Subrange Types

A subrange type defines a range of values within the range of another type (hence the name *subrange*). For example, you can define a subrange of the Integer type, from 1 to 10 or from 100 to 1000, or you can define a subrange of the Char type with English uppercase characters only, as in:

```
type
  TTen = 1..10;
  TOverHundred = 100..1000;
  TUppercase = 'A'..'Z';
```

In the definition of a subrange, you don't need to specify the name of the base type. You just need to supply two constants of that type. The original type must be an ordinal type, and the resulting type will be another ordinal type. When you have defined a variable as a subrange, you can then assign it any value within that range. This code is valid:

```
var
  UppLetter: TupperCase;

begin
  UppLetter := 'F';
```

But this is not:

```
var
  UppLetter: TupperCase;

begin
  UppLetter := 'e'; // compile-time error
```

Writing the code above results in a compile-time error, *"Constant expression violates subrange bounds."* If you write the following code instead, the compiler will accept it:

```
var
  UppLetter: Tuppercase;
  Letter: Char;
```

76 - 02: Variables and Data Types

```
begin
  Letter := 'e';
  UpLetter := Letter;
```

At run-time, if you have enabled the Range Checking compiler option (in the Compiler page of the Project Options dialog box), you'll get a *Range check error* message, as expected. This is similar to the integer type overflow errors which I described earlier.

I suggest that you turn on this compiler option while you are developing a program, so it'll be more robust and easier to debug, as in case of errors you'll get an explicit message and not an undetermined behavior. You can eventually disable this option for the final build of the program, so that it will run a little faster. However, the increase in speed is almost negligible so I suggest to leave all of these run-time checks turned on, even in a shipping program.

Enumerated Types

Enumerated types (usually referred to as “enums”) constitute another user-defined ordinal type. Instead of indicating a range of an existing type, in an enumeration you list all of the possible values for the type. In other words, an enumeration is a list of (constant) values. Here are some examples:

```
type
  TColors = (Red, Yellow, Green, Cyan, Blue, Violet);
  TSuit = (Club, Diamond, Heart, Spade);
```

Each value in the list has an associated *ordinality*, starting with zero. When you apply the `Ord` function to a value of an enumerated type, you get this “zero-based” value. For example, `Ord (Diamond)` returns 1.

Enumerated types can have different internal representations. By default, Delphi uses an 8-bit representation, unless there are more than 256 different values, in which case it uses the 16-bit representation. There is also a 32-bit representation, which at times is useful for compatibility with C or C++ libraries.

note You can change the default representation of enumerated types, asking for a larger one regardless of the number of elements in the enumeration, by using the `$Z` compiler directive. This is a fairly rare setting.

Scoped Enumerators

The specific constant values of an enumerated type can be considered to all effects as global constants, and there have been cases of names conflicts among different

enumerated values. This is why the language supports scoped enumerations, a feature you can activate using a specific compiler directive, `$SCOPEDENUMS`, and which requires you to refer to the enumerated value using the type name as a prefix:

```
// classic enumerated value
S1 := Club;

// "scoped" enumerated value
S1 := TSuit.Club;
```

When this feature was introduced, the default coding style remained the traditional behavior, to avoid breaking existing code. Scoped enumerators, in fact, change the behavior of enumerations making it compulsory to refer to them with a fully qualified type prefix.

Having an *absolute* name to refer to enumerated values removes the risk of a conflict, could let you avoid using the initial prefix of the enumerated values as a way to differentiate with other enumerations, and makes the code more readable, even if much longer to write.

As an example, the `System.IOUtils` unit defines this type:

```
{ $SCOPEDENUMS ON }
type
  TSearchOption = (soTopDirectoryOnly, soAllDirectories);
```

This means you cannot refer to the second value as `soAllDirectories`, but you have to refer to the enumerated value with its complete name:

```
TSearchOption.soAllDirectories
```

The FireMonkey platform library uses quite a number of scoped enumerators, as well, requiring the type as a prefix to the actual values, while the older VCL library is generally based on the more traditional model. The RTL is a mix of the two.

note Enumerated values in Object Pascal libraries often use two or three initials of the type at the beginning of the value, using lowercase letters by convention, like “so” for Search Options in the example above. When using the type as a prefix, this might seem a bit redundant, but given the commonality of the approach, I don't see it going away any time soon.

Set Types

Set types indicate a group of values, where the list of available values is indicated by the ordinal type the set is based onto. These ordinal types are usually limited, and quite often represented by an enumeration or a subrange.

78 - 02: Variables and Data Types

If we take the subrange 1..3, the possible values of the set based on it include only 1, only 2, only 3, both 1 and 2, both 1 and 3, both 2 and 3, all the three values, or none of them.

A variable usually holds one of the possible values of the range of its type. A set-type variable, instead, can contain none, one, two, three, or more values of the range. It can even include all of the values.

Here is an example of a set:

```
type
  TSuit = (Club, Diamond, Heart, Spade);
  TSuits = set of TSuit;
```

Now I can define a variable of this type and assign to it some values of the original type. To indicate some values in a set, you write a comma-separated list, enclosed within square brackets. The following code shows the assignment to a variable of several values, a single value, and an empty value:

```
var
  Cards1, Cards2, Cards3: TSuits;

begin
  Cards1 := [Club, Diamond, Heart];
  Cards2 := [Diamond];
  Cards3 := [];
```

In Object Pascal, a set is generally used to indicate several nonexclusive flags. For example a value based on a set type is the style of a font. Possible values indicate a bold, italic, underline, and strike-through font. Of course the same font can be both italic and bold, have no attributes, or have them all. For this reason it is declared as a set. You can assign values to this set in the code of a program as follows:

```
Font.Style := []; // no style
Font.Style := [fsBold]; // bold style only
Font.Style := [fsBold, fsItalic]; // two styles active
```

Set Operators

We have seen that sets are a very Pascal-specific user defined data type. That's why the set operators are worth a specific coverage. They include union (+), difference (-), intersection (*), membership test (in), plus some relational operators.

To add an element to a set, you can make the union of the set with another one that has only the elements you need. Here's an example related to font styles:

```
// add bold
Style := Style + [fsBold];

// add bold and italic, but remove underline if present
Style := Style + [fsBold, fsItalic] - [fsUnderline];
```

As an alternative, you can use the standard `Include` and `Exclude` procedures, which are much more efficient (but cannot be used with component properties of the set type):

```
Include (Style, fsBold);
Exclude (Style, fsItalic);
```

Expressions and Operators

We have seen that you can assign to a variable a type-compatible literal value, a constant value, or the value of another variable. In many cases, what you assign to a variable is the result of an expression, involving one or more values and one or more operators. Expressions are another core element of the language.

Using Operators

There isn't a general rule for building expressions, since they mainly depend on the operators being used, and Object Pascal has a number of operators. There are logical, arithmetic, Boolean, relational, and set operators, plus some other special ones:

```
// sample expressions
20 * 5      // multiplication
30 + n      // addition
a < b       // less than comparison
-4          // negative value
c = 10      // test for equality (like == in C syntax)
```

Expressions are common to most programming languages, and most operators are the same. An expression is any valid combination of constants, variables, literal values, operators, and function results. Expressions can be used to determine the value to assign to a variable, to compute the parameter of a function or procedure, or to test for a condition. Every time you are performing an operation on the value of an identifier, rather than using an identifier by itself, you are using an expression.

note The result of an expression is generally stored in a temporary variable of the proper data type automatically generated by the compiler on your behalf. You might want to use an explicit variable when you need to compute the same expression more than once in the same code fragment. Notice that complex expressions might require multiple temporary variables to store intermediate results, again something the compiler takes care of for you and you can generally ignore.

Showing the Result of an Expression

If you want to make a few experiments with expressions, there is nothing better than writing a simple program. As for most of the initial demos of this book, create a simple program based on a form, and use the custom `Show` function to display something to the user. In case the information you want to show is not a string message but number or a Boolean logical value, you need to convert it, for example calling the `IntToStr` or `BoolToStr` function.

note In Object Pascal parameters passed to a function or procedures are enclosed in parenthesis. Some other languages (notably Rebol and, to some extent, Ruby) let you pass parameters simply by writing them after the function or procedure name. Getting back to Object Pascal, nested functions calls use nested parenthesis, like in the code below.

Here is a sample code snippet from the `ExpressionsTest` application project (in which I'm using the classic `IntToStr` syntax for clarity, given the parameter is an expression:

```
Show (IntToStr (20 * 5));  
Show (IntToStr (30 + 222));  
Show (BoolToStr (3 < 30, True));  
Show (BoolToStr (12 = 10, True));
```

The output is this code snippet is quite trivial:

```
100  
252  
True  
False
```

I've provided this demo as a skeleton for you to try out different types of expressions and operators, and see the corresponding output.

note Expressions you write in Object Pascal are parsed by the compiler and generate assembly code. If you want to change one of these expressions, you need to change the source code and recompile the application. The system libraries, however, have support for dynamic expressions calculated at run time, a feature tied to reflection and covered in Chapter 16.

Operators and Precedence

Expressions are made of operators applied to values. As I mentioned, most operators are shared among the various programming languages and are quite intuitive, such as the basic match and comparison operators. In this section I'll highlight only specific elements of Object Pascal operators.

You can see a list of the operators of the language below, grouped by precedence and compared to operators in C#, Java, and Objective-C (and most languages based on the C language syntax, anyway).

Relational and Comparison Operators (Lowest Precedence)

=	Test whether equal (in C this is ==)
<>	Test whether not equal (in C this is !=)
<	Test whether less than
>	Test whether greater than
<=	Test whether less than or equal to, or a subset of a set
>=	Test whether greater than or equal to, or a superset of a set
in	Test whether the item is a member of the set
is	Test whether an object is compatible with a given type (covered in Chapter 8) or implements a given interface (covered in Chapter 11)

Additive Operators

+	Arithmetic addition, set union, string concatenation, pointer offset addition
-	Arithmetic subtraction, set difference, pointer offset subtraction
or	Boolean or bitwise or (in C this is either or)
xor	Boolean or bitwise exclusive or (in C bitwise exclusive or is ^)

Multiplicative and Bitwise Operators

*	Arithmetic multiplication or set intersection
/	Floating-point division
div	Integer division (in C this also uses /)
mod	Modulo (the remainder an of integer division) (in C this is %)
as	Allows a type-checked conversion at runtime (covered in Chapter 8)
and	Boolean or bitwise and (in C this is either && or &)
shl	Bitwise left shift (in C this is <<)
shr	Bitwise right shift (in C this is >>)

Unary Operators (Highest Precedence)

@	Memory address of a variable or function (returns a pointer, in C this is &)
not	Boolean or bitwise not (in C this is !)

82 - 02: Variables and Data Types

Differently from many other programming languages, the logical operators (including `and`, `or`, and `not`) have a higher precedence than the comparison operators (including `less than` and `greater than`). So if you write:

```
| a < b and c < d
```

the compiler will do the `and` operation first, generally resulting in a type compatibility compiler error in the expression. If you want to test both comparisons, you should enclose each of the *less than* expressions in parentheses:

```
| (a < b) and (c < d)
```

For math operations, instead, the common rules apply, with multiplication and division taking precedence over addition and subtraction. The first two expressions below are equivalent, while the third is different:

```
| 10 + 2 * 5      // result is 20
| 10 + (2 * 5)    // result is 20
| (10 + 2) * 5    // result is 60
```

tip While in some cases parenthesis are not needed, given you can count on the language operator precedence rules, it is highly recommended to add them anyway as these rules vary depending on programming language and it's always better to be more clear for anyone reading or modifying the code in the future.

Some of the operators have different meanings when used with different data types. For example, the `+` operator can be used to add two numbers, concatenate two strings, make the union of two sets, and even add an offset to a pointer (if the specific pointer type has *pointer math* enabled):

```
| 10 + 2 + 11
| 10.3 + 3.4
| 'Hello' + ' ' + 'world'
```

However, you cannot add two characters, as is possible in C.

An unusual operator is `div`. In Object Pascal, you can divide any two numbers (real or integers) with the `/` operator, and you'll invariably get a real-number result. If you need to divide two integers and want an integer result, use the `div` operator instead. Here are two sample assignments (this code will become clearer as we cover data types in the next chapter):

```
| realValue := 123 / 12;
| integerValue := 123 div 12;
```

To make sure the integer division has no remainder, you can use the `mod` operator and check if the result is zero, like in the following Boolean expression:

```
| (x mod 12) = 0
```

Date and Time

While there was no native type for date and time in the early versions of the Pascal language, Object Pascal has a native type for date and time. It uses a floating-point representation to handle date and time information. To be more specific the `System` unit defines a specific `TDateTime` data type for that purpose.

This is a floating-point type, because it must be wide enough to store years, months, days, hours, minutes, and seconds, down to millisecond resolution in a single variable:

- Dates are stored as the number of days since 1899-12-30 (with negative values indicating dates before 1899) in the integer part of the `TDateTime` value
- Times are stored as fractions of a day in the decimal part of the value

history In case you are wondering where that strange date comes from, there is a rather long story behind it tied to Excel and dates representations in Windows applications. The idea was to consider day number 1 as the first of January 1900, so that New Year's eve of 1899 would have been day number 0. However, the original developer of that date representation duly forgot that year 1900 wasn't a leap year, and so calculations were later adjusted by 1 day, turning the first of January 1900 into day number 2.

As mentioned, `TDateTime` is not a predefined type the compiler understands, but it is defined in the `System` unit as:

```
type
  TDateTime = type Double;
```

note The `System` unit could be somehow considered almost as part of the core language, given it is always automatically included in each compilation, without a `uses` statement (actually adding the `System` unit to a `uses` section will cause a compilation error). Technically, though, this unit is considered as the core part of the run-time library (RTL), and it will be covered in Chapter 17.

There are also two related types to handle the time and date portions of a `TDateTime` structure, defined as `TDate` and `TTime`. These specific types are aliases of the full `TDateTime`, but they are treated by system functions trimming the unused part of the data.

Using date and time data types is quite easy, because Delphi includes a number of functions that operate on this type. There are several core functions in the `System.SysUtils` unit, and many specific functions in the `System.DateUtils` unit (which despite the name includes also functions for manipulating time).

84 - 02: Variables and Data Types

Here you can find a short list of commonly used date/time manipulation functions:

<code>Now</code>	Returns the current date and time into a date/time value.
<code>Date</code>	Returns only the current date.
<code>Time</code>	Returns only the current time.
<code>DateTimeToStr</code>	Converts a date and time value into a string, using default formatting; to have more control on the conversion use the <code>FormatDateTime</code> function instead.
<code>DateToStr</code>	Converts the date portion of a date/time value into a string.
<code>TimeToStr</code>	Converts the time portion of a date/time value into a string.
<code>FormatDateTime</code>	Formats a date and time using the specified format; you can specify which values you want to see and which format to use by providing a complex format string.
<code>StrToDateTime</code>	Converts a string with date and time information to a date/time value, raising an exception in case of an error in the format of the string. Its companion function, <code>StrToDateTimeDef</code> returns the default value in case of an error rather than raising an exception.
<code>DayOfWeek</code>	Returns the number corresponding to the day of the week of the date/time value passed as parameter (using locale configuration).
<code>DecodeDate</code>	Retrieves the year, month, and day values from a date value.
<code>DecodeTime</code>	Retrieves the hours, minutes, seconds, and milliseconds from a date value.
<code>EncodeDate</code>	Turns year, month, and day values into a date/time value.
<code>EncodeTime</code>	Turns hour, minute, second, and millisecond values into a date/time value.

To show you how to use this data type and some of its related routines, I've built a simple application project, named `TimeNow`. When the program starts it automatically computes and displays the current time and date.

```
var
  StartTime: TDateTime;
begin
  StartTime := Now;
  Show ('Time is ' + TimeToStr (StartTime));
  Show ('Date is ' + DateToStr (StartTime));
```

The first statement is a call to the `Now` function, which returns the current date and time. This value is stored in the `StartTime` variable.

note When an Object Pascal function is called with no parameters, there is no need to type the empty parentheses unlike with the C style languages.

The next two statements display the time portion of the `TDateTime` value, converted into a string, and the date portion of the same value. This is the output of the program (which will depend on your locale configuration):

```
| Time is 6:33:14 PM
| Date is 10/7/2020
```

To compile this program you need to refer to functions that are part of the unit `System.SysUtils` (a short name for “system utilities”). Besides calling `TimeToStr` and `DateToStr` you can use the more powerful `FormatDateTime` function.

Notice that time and date values are transformed into strings depending on the system's international settings. The date and time formatting information is read from the system, depending on the operating system and the locale, populating a `TFormatSettings` data structure. If you need customized formatting, you can create a custom structure of that type and pass it as parameter to most date time formatting functions.

note The `TimeNow` project has also a second button you can use to enable a timer. This is a component that executes an event handler automatically over time (you specify the interval). In the demo, if you enable the timer you'll see the current time added to the list every second. A more useful user interface would be to update a label with the current time every second, basically building a clock.

Typecasting and Type Conversions

As we have seen, you cannot assign a variable of one data type to one of a different type. The reason is, depending on the actual representation of the data, you might end up with something meaningless.

Now, this is not true for each and every data type. Numerical types, for example, can always be promoted safely. “Promoted” here means you can always safely assign a value to a type with a larger representation. So you can assign a word to an integer, and an integer to an `Int64` value. The opposite operation, called “demotion”, is allowed by the compiler but it will issue a warning, because you might end up with partial data. Other automatic conversions are one way only: For example, you can assign an integer to a floating point number, but the opposite operation is illegal.

There are scenarios you want to change the type of a value and the operation makes sense. When you need to do this, there are two choices. One is to perform a direct type cast, which will copy the physical data and might result in a proper conversion or a not depending on the types. When you perform a typecast, you are telling the

86 - 02: Variables and Data Types

compiler “I know what I'm doing, let me go for it”. If you use type casts but aren't really sure what you are doing, you can get in trouble as you lose the compiler type-checking safety net.

Type casting uses a simple functional notation, with the name of the destination data type used as a function:

```
var
  I: Integer;
  C: Char;
  B: Boolean;

begin
  I := Integer ('X');
  C := Char (I);
  B := Boolean (I);
```

You can safely typecast between data types having the same size (that is the same number of bytes to represent the data – unlike in the code snippet above!). It is usually safe to typecast between ordinal types, but you can also typecast between pointer types (and also objects) as long as you know what you are doing.

Direct type casting is a dangerous programming practice, because it allows you to access a value as if it represented something else. Since the internal representations of data types generally do not match (and might even change depending on the target platform), you risk accidentally creating hard-to-track errors. For this reason, *you should generally avoid type casting*.

The second choice to assign a variable to one of a different type is to use a type conversion function. A list of functions allowing you to convert between various basic types are summarized below (and I've already used some of these functions in the demos of this chapter):

Chr	Converts an ordinal number into a character.
Ord	Converts an ordinal-type value into the number indicating its order.
Round	Converts a real-type value into an Integer-type value, rounding its value (also see the following note).
Trunc	Converts a real-type value into an Integer-type value, truncating its value.
Int	Returns the Integer part of the floating-point value argument.
FloatToDecimal	Converts a floating-point value to record including its decimal representation (exponent, digits, sign).
FloatToStr	Converts a floating-point value to its string representation using default formatting.
StrToFloat	Converts a string to a floating-point value.

note The implementation of the `Round` function is based on the native implementation offered by the CPU. Modern processors generally adopt the so-called "Banker's Rounding", which rounds middle values (such as 5.5 or 6.5) up and down depending whether they follow an odd or an even number. There are other rounding functions, such as `RoundTo`, that offer you more control on the actual operation.

As mentioned earlier in this chapter, some of these conversion functions are also available as direct operations on the data type (thanks to the type helper mechanism). While there are classic conversions like `IntToStr`, you can apply the `ToString` operation to most numeric types to convert them to a string representation. There are many conversions you can apply directly to variables using type helpers, and that should be your preferred coding style.

Some of these routines work on the data types that we'll discuss in the following sections. Notice that the table doesn't include routines for special types (such as `TDateTime` or `Variant`) or routines specifically intended for formatting more than conversion, like the powerful `Format` and `FormatFloat` routines.

88 - 02: Variables and Data Types

03: language statements

If the concept of data type was one of the breakthrough of the Pascal programming language when it was first invented, the other side is represented by the code or programming statements. At that time, this idea was clarified by Nicklaus Wirth's outstanding book “Algorithms + Data Structures = Programs”, published by Prentice Hall in February 1976 (a classic book, still reprinted and available). While this book predates object-oriented programming by many years, it can be considered one of the foundations of modern programming, based on a strong notion of data type, and in this way a foundation of the concepts that lead to object-oriented programming languages.

Statements of the programming language are based on keywords and other elements which allow you to indicate to a compiler a sequence of operations to perform. Statements are often enclosed in procedures or functions, as we'll start to see in more detail in the next chapter. For now, we'll just focus on the basic types of instructions you can write to create a program. As we saw in Chapter 1 (in the section covering white space and code formatting), the actual program code can be written quite freely. I also covered comments and some other special elements, but never fully introduced some core concepts, like a programming statement.

Simple and Compound Statements

Programming instructions are generally called *statements*. A program block can be made of *several* statements. There are two types of statements, simple and compound.

A statement is called *simple* when it doesn't contain any other sub-statements. Examples of simple statements are assignment statements and procedure calls. In Object Pascal simple statements are *separated* by a semicolon:

```
X := Y + Z;    // assignment
Randomize;    // procedure call
...
```

To define a *compound* statement, you can include one or more statements within the keywords `begin` and `end`, which act as containers of a multiple statements and have a role similar, but not identical, to curly braces in C-derived languages. A compound statement can appear anywhere a simple Object Pascal statement can appear. Here is an example:

```
begin
  A := B;
  C := A * 2;
end;
```

The semicolon after the last statement of the compound statement (that is, before the `end`) isn't required, as in the following:

```
begin
  A := B;
  C := A * 2
end;
```

Both versions are correct. The first version has a useless (but harmless) final semicolon. This semicolon is, in fact, a null statement or an empty statement; that is, a statement with no code. This is significantly different from many other programming languages (like those based on the C syntax), in which the semicolon is a statement *terminator* (not a separator) and is always required at the end of a statement.

Notice that, at times, a null statement can be specifically used inside loops or in other particular cases in place of an actual statement, as in:

```
while condition_with_side_effect do
  ; // null or empty statement
```

Although these final semicolons serve no purpose, most developers tend to use them and I suggest you to do the same. Sometimes after you've written a couple of lines you might want to add one more statement. If the last semicolon is missing

you have to remember to add it, so it is usually better to add it in the first place. As we'll see right away, there is an exception to this rule of adding extra semicolons, and that is when the next element is an `else` statement inside a condition.

The If Statement

A conditional statement is used to execute either one of the statements it contains or none of them, depending on a specific test (or condition). There are two basic flavors of conditional statements: `if` statements and `case` statements.

The `if` statement can be used to execute a statement only if a certain condition is met (`if-then`) or to choose between two different alternatives (`if-then-else`). The condition is defined with a Boolean expression.

A simple Object Pascal example, called `IfTest`, will demonstrate how to write conditional statements. In this program we'll use a checkbox to get user input, by reading its `IsChecked` property (and storing it to a temporary variable, although this isn't strictly required, as you could directly check the property value in the conditional expression):

```
var
  IsChecked: Boolean;
begin
  IsChecked := CheckBox1.IsChecked;
  if IsChecked then
    Show ('checkbox is checked');
```

If the checkbox is checked, the program will show a simple message. Otherwise nothing happens. By comparison, the same statement using the C language syntax will look like the following (where the conditional expression *must* be enclosed within parentheses):

```
if (IsChecked)
  Show ("checkbox is checked");
```

Some other languages have the notion of an `endif` element to allow you to write multiple statements, where in Object Pascal syntax the conditional statement is a single statement by default. You use a `begin-end` block to execute more than one statement as part of the same condition.

If you want to do different operations depending on the condition, you can use an `if-then-else` statement (and in this case I used a direct expression to read the check box status):

```
// if-then-else statement
```

92 - 03: Language Statements

```
if CheckBox1.IsChecked then
    Show ('checkbox is checked')
else
    Show ('checkbox is not checked');
```

Notice that you cannot have a semicolon after the first statement and before the `else` keyword or the compiler will issue a syntax error. The reason is that the `if-then-else` statement is a single statement, so you cannot place a semicolon in the middle of it.

An `if` statement can be quite complex. The condition can be turned into a series of conditions (using the `and`, `or`, and `not` Boolean operators), or the `if` statement can nest a second `if` statement. Beside nesting `if` statements, when there are multiple distinct conditions, it is common to have consecutive statements `if-then-else-if-then`. You can keep chaining as many of these `else-if` conditions as you want.

The third button of the `IfTest` application project demonstrates these scenarios, using the first character of an edit box (which might be missing, hence the external test) as input:

```
var
    AChar: Char;
begin
    // multiple nested if statements
    if Edit1.Text.Length > 0 then
        begin
            AChar := Edit1.Text.Chars[0];

            // checks for a lowercase char (two conditions)
            if (AChar >= 'a') and (AChar <= 'z') then
                Show ('char is lowercase');

            // follow up conditions
            if AChar <= Char(47) then
                Show ('char is lower symbol')
            else if (AChar >= '0') and (AChar <= '9') then
                Show ('char is a number')
            else
                Show ('char is not a number or lower symbol');
        end;
```

Look at the code carefully and run the program to see if you understand it (and play with similar programs you can write to learn more). You can consider more options and Boolean expressions and increase the complexity of this small example, making any test you like.

Case Statements

If your `if` statements become very complex and they are based on tests for ordinal values, you can consider replacing them with case statements. A case statement consists of an expression used to select a value and a list of possible values, or a range of values. These values are constants, and they must be unique and of an ordinal type. Eventually, there can be an `else` statement that is executed if none of the values you specified correspond to the value of the selector. While there isn't a specific `endcase` statement, a case is always terminated by an `end` (which in this case isn't a block terminator, as there isn't a matching `begin`).

note Creating a case statement requires an ordinal value. A case statement based on a string value is currently not allowed. In that case you need to use nested `if` statements or a different data structure, like a dictionary (as I show later in the book in Chapter 14).

Here is an example (part of the `CaseTest` project), which uses as input the integral part of the number entered in a `NumberBox` control, a numeric input control:

```
var
  Number: Integer;
  AText: string;
begin
  Number := Trunc(NumberBox1.Value);
  case Number of
    1: AText := 'One';
    2: AText := 'Two';
    3: AText := 'Three';
  end;
  if AText <> '' then
    Show(AText);
```

Another example is the extension of the previous complex `if` statement, turned into a number of different conditions of a case test:

```
case AChar of
  '+' : AText := 'Plus sign';
  '-' : AText := 'Minus sign';
  '*', '/': AText := 'Multiplication or division';
  '0'..'9': AText := 'Number';
  'a'..'z': AText := 'Lowercase character';
  'A'..'Z': AText := 'Uppercase character';
  #12032..#12255: AText := 'Kangxi Radical';
else
  AText := 'Other character: ' + aChar;
end;
```

94 - 03: Language Statements

note As you can see in the previous code snippet, a range of values is defined with the same syntax of a subrange data type. Multiple values for a single branch, instead, are separated by a comma. For the Kangxi Radical section I've used the numerical value rather than the actual characters, because most of the fixed-size fonts used by the IDE editor won't display the symbols properly.

It is considered good practice to include the `else` part to signal an undefined or unexpected condition. A `case` statement in Object Pascal selects one execution path, it doesn't position itself at an entry point. In other word, it will execute the statement or block after the colon of the selected value and it will skip to the next statement after the case.

This is very different from the C language (and some of its derived languages) which treat branches of a `switch` statement as entry points and will execute all following statements unless you specifically use a `break` request (although this is a specific scenario in which Java and C# actually differ in their implementation). The C language syntax is like the following:

```
switch (aChar) {
    case '+': aText = "plus sign"; break;
    case '-': aText = "minus sign"; break;
    ...
    default: aText = "unknown"; break;
}
```

The For Loop

The Object Pascal language has the typical repetitive or looping statements of most programming languages, including `for`, `while`, and `repeat` statements, plus the more modern `for-in` (or *for-each*) cycle. Most of these loops will be familiar if you've used other programming languages, so I'll only cover them briefly (indicating the key differences from other languages).

The `for` loop in Object Pascal is strictly based on a counter, which can be either increased or decreased each time the loop is executed. Here is a simple example of a `for` loop used to add the first ten numbers (part of the `ForTest` demo).

```
var
    Total, I: Integer;
begin
    Total := 0;
    for I := 1 to 10 do
        Total := Total + I;
    Show(Total.ToString);
```

For those curious, the output is 55. Another way you can write a for loop after the introduction of inline variables is to declare the loop counter variable within the declaration (with a syntax that somehow resembled for loops in C and derived languages discussed later):

```
for var I: Integer := 1 to 10 do
    Total := Total + I;
```

In this case, you can also take advantage of type inference and omit the type specification. The complete code snippet above becomes:

```
var
    Total: Integer;
begin
    Total := 0;
    for var I := 1 to 10 do
        Total := Total + I;
    Show(Total.ToString);
```

One advantage of using an inline loop counter is that the scope of that variable will be limited to the loop: Using it after the for statement will cause an error, while in general you get only a warning when using the loop counter outside of the loop.

The for loop in Pascal is less flexible than in other languages (it is not possible to specify an increment different than one), but it is simple and easy to understand. As a comparison, this is the same for loop written in the C language syntax:

```
int total = 0;
for (int i = 1; i <= 10; i++) {
    total = total + i;
}
```

In these languages, the increment is an expression that can specify any kind of sequence, which can lead code many would consider unreadable, as the following:

```
int total = 0;
for (int i = 10; i > 0; total += i--) {
    ...
}
```

In Object Pascal, instead, you can only use a single step increment. If you want to test for a more complex condition, or if you want to provide a customized counter, you'll need to use a while or repeat statement, instead of a for loop.

The only alternative to single increment is single decrement, or a reverse for loop with the `downto` keyword:

```
var
    Total, I: Integer;
begin
    Total := 0;
    for I := 10 downto 1 do
        Total := Total + I;
```


96 - 03: Language Statements

note Reverse counting is useful, for example, when you are affecting a list-based data structure you are looping through. When deleting some elements, you often go backwards, as with a forward loop you might affect the sequence you are operating on (that is, if you delete the third element of a list, the fourth element becomes the third: now you are on the third, move to the next one (the fourth) but you are actually operating on what was the fifth element, skipping one).

In Object Pascal the counter of a `for` loop doesn't need to be a number. It can be a value of any ordinal type, such as a character or an enumerated type. This helps you write more readable code. Here is an example with a `for` loop based on the `Char` type:

```
var
  AChar: Char;
begin
  for AChar := 'a' to 'z' do
    Show (AChar);
```

This code (part of the `ForTest` program) shows all of the letters of the English alphabet, each one in a separate line of the output Memo control.

note I've already shown a similar demo, but based on an integer counter, as part of the `CharsTest` example of Chapter 2. In that case, though, the chars were concatenated in a single output string.

Here is another code snippet that shows a `for` loop based on a custom enumeration:

```
type
  TSuit = (Club, Diamond, Heart, Spade);
var
  ASuit: TSuit;
begin
  for ASuit := Club to Spade do
    ...
```

This last loop that cycles on all of the elements of the data type. It could be better written to explicitly operate on each element of the type (making it more flexible to changes in the definition) rather than specifically indicating the first and the last element, by writing:

```
for ASuit := Low (TSuit) to High (TSuit) do
```

In a similar way, it is quite common to write `for` loop on all elements of a data structure, such as a string. In this case you can use this code (from the `ForTest` project):

```
var
  S: string;
  I: Integer;
begin
  S := 'Hello world';
  for I := Low (S) to High (S) do
    Show(S[I]);
```

If you prefer not to have to indicate the first and last element of the data structure, you can use a `for-in` loop instead, a special-purpose `for` loop discussed in the next section.

note How the compiler treats direct read of the string data using the `[]` operators and determines the lower and upper bounds of a string remains a rather complex topic in Object Pascal, even if defaults are now the same for all platforms. This will be covered in Chapter 6.

For data structures that use zero-based indexing, you'd want to loop from index zero to the one before the data structure size or length. Common ways of writing this code are:

```
for I := 0 to Count-1 do ...
for I := 0 to Pred(Count) do ...
```

One final note regarding `for` loops is what happens to the loop counter after the loop. In short, the value is *unspecified* and the compiler will issue a warning if you try to use the `for` loop counter after the loop has terminated. One advantage of using an inline variable for the loop counter is that the variable is defined only within the loop itself and it won't be accessible after its end statement, resulting in a compiler error (which is a stronger protection):

```
begin
  var Total := 0;
  for var I: Integer := 1 to 10 do
    Inc (Total, I);
  Show (Total.ToString);
  Show (I.ToString); // compiler error: Undeclared Identifier 'I'
```

The for-in Loop

Object Pascal has a specific loop construct for cycling over all of the elements of a list or collection, called `for-in` (a feature other programming languages call *for each*). In this `for` loop the cycle operates on each element of an array, a list, a string, or some other type of container. Unlike C#, Object Pascal doesn't require implementing the `IEnumerator` interface, but the internal implementation is somewhat similar.

note You can find the technical details of how to support the `for-in` loop in a class, adding custom enumeration support, in Chapter 10.

Let's start with a very simple container, a string, which can be seen as a collection of characters. We have seen at the end of the previous section how to use a `for` loop to

98 - 03: Language Statements

operate on all elements of a string. The same exact effect can be obtained with the following `for-in` loop based on a string, where the `ch` variable receives as value each of the string elements in turn:

```
var
  S: string;
  ch: Char;
begin
  S := 'Hello world';
  for ch in S do
    Show(ch);
```

This snippet is also part of the `ForTest` application project. The advantage over using a traditional `for` loop is that you don't need to remember which is the first element of the string and how to extract the position of the last one. This loop is easier to write and maintain and has a similar efficiency.

Like traditional `for` loops, also `for-in` loops can benefit from using an inline variable declaration. We can rewrite the code above using the following equivalent code:

```
var
  S: string;
begin
  S := 'Hello world';
  for var ch: Char in S do
    Show(ch);
```

The `for-in` loop can be used to access the elements of the several different data structures:

- Characters in a string (see the previous code snippet)
- Active values in a set
- Items in a static or dynamic array, including two-dimensional arrays (covered in Chapter 5)
- Objects referenced by classes with `GetEnumerator` support, including many predefined ones like strings in a string list, elements of the various container classes, the components owned by a form, and many others. How to implement this will be discussed in Chapter 10.

Now it is a little difficult at this point in the book to cover these advanced usage patterns, so I'll get back to examples of this loop later in the book.

note The `for-in` loop in some languages (for example JavaScript) has a bad reputation for being very slow to run. This is not the case in Object Pascal, where it takes about the same time of a corresponding standard `for` loop. To prove this, I've added to the `LoopsTest` application project some timing code, which first creates a string of 30 million elements and later scans it with both types of loops (doing a very simple operation at each iteration. The difference in speed is about 10% in favor of the classic `for` loop (62 milliseconds vs. 68 milliseconds on my Windows machine).

While and Repeat Statements

The idea behind the `while-do` and the `repeat-until` loops is repeating the execution of a code block over and over until a given condition is met. The difference between these two loops is that condition is checked at the beginning or at the end of the loop. In other words, the code block of the `repeat` statement is always executed at least once.

note Most other programming languages have only one type of open looping statement, generally called and behaving like a `while` loop. The C language syntax has the same two options as the Pascal syntax, with the `while` and `do-while` cycles. Notice, though, that they use the same logical condition, differently from the `repeat-until` loop that has a reverse condition.

You can easily understand why the `repeat` loop is always executed at least once, by looking at a simple code example:

```
while (I <= 100) and (J <= 100) do
  begin
    // use I and J to compute something...
    I := I + 1;
    J := J + 1;
  end;

repeat
  // use I and J to compute something...
  I := I + 1;
  J := J + 1;
until (I > 100) or (J > 100);
```

note You will have noticed that in both the `while` and `repeat` conditions I have enclosed the “sub-conditions” in parentheses. It is necessary in this case, as the compiler will execute `or` before performing the comparisons (as I covered in the section about operators of Chapter 2).

If the initial value of `I` or `J` is greater than 100, the `while` loop is completely skipped, while statements inside the `repeat` loop are executed once anyway.

100 - 03: Language Statements

The other key difference between these two loops is that the `repeat-until` loop has a *reversed* condition. This loop is executed *as long as the condition is not met*. When the condition is met, the loop terminates. This is the opposite of a `while-do` loop, which is executed while the condition is true. For this reason I had to reverse the condition in the code above to obtain a similar effect.

note The “reverse condition” is formally known as the “De Morgan's” laws (described, for example, on Wikipedia at http://en.wikipedia.org/wiki/De_Morgan%27s_laws).

Examples of Loops

To explore some more details of loops, let's look at a small practical example. The `LoopsTest` program highlights the difference between a loop with a fixed counter and a loop with an open counter. The first fixed counter loop, a `for` loop, displays numbers in sequence:

```
var
  I: Integer;
begin
  for I := 1 to 20 do
    Show ('Number ' + IntToStr (I));
  end;
```

The same could have been obtained also with a `while` loop, with an internal increment of one (notice you increment the value after using the current one). With a `while` loop, however, you are free to set a custom increment, for example by 2:

```
var
  I: Integer;
begin
  I := 1;
  while I <= 20 do
    begin
      Show ('Number ' + IntToStr (I));
      Inc (I, 2)
    end;
  end;
```

This code shows all of the odd numbers from one to 19. These loops with fixed increments are logically equivalent and execute a predefined number of times. This is not always the case. There are loops that are more undetermined in their execution, depending for example on external conditions.

note When writing a `while` loop you must always consider the case where the condition is never met. For example, if you write the loop above but forget to increment the loop counter, this will result into an infinite loop (which will stall the program forever, consuming the CPU at 100%, until the operating system kills it).

To show an example of a less deterministic loop I've written a `while` loop still based on a counter, but one that is increased randomly. To accomplish this, I've called the `Random` function with a range value of 100. The result of this function is a number between 0 and 99, chosen randomly. The series of random numbers control how many times the `while` loop is executed:

```
var
  I: Integer;
begin
  Randomize;
  I := 1;
  while I < 500 do
  begin
    Show ('Random Number: ' + IntToStr (I));
    I := I + Random (100);
  end;
end;
```

If you remember to add a call the `Randomize` procedure, which resets the random number generator at a different point for each program execution, each time you run the program, the numbers will be different. The following is the output of two separate executions, displayed side by side:

Random Number: 1	Random Number: 1
Random Number: 40	Random Number: 47
Random Number: 60	Random Number: 104
Random Number: 89	Random Number: 201
Random Number: 146	Random Number: 223
Random Number: 198	Random Number: 258
Random Number: 223	Random Number: 322
Random Number: 251	Random Number: 349
Random Number: 263	Random Number: 444
Random Number: 303	Random Number: 466
Random Number: 349	
Random Number: 366	
Random Number: 443	
Random Number: 489	

Notice that not only are the generated numbers different each time, but so is the number of items. This `while` loop is executed a random numbers of times. If you execute the program several times in a row, you'll see that the output has a different number of lines.

Breaking the Flow with Break and Continue

Despite the differences, each of the loops lets you execute a block of statements a number of times, based on some rules. However, there are scenarios you might want to add some additional behavior. Suppose, as an example, you have a for loop where you search for the occurrence of a given letter (this code is part of the FlowTest application project):

```
var
  S: string;
  I: Integer;
  Found: Boolean;
begin
  S := 'Hello world';
  Found := False;
  for I := Low (S) to High (S) do
    if (S[I]) = 'o' then
      Found := True;
```

At the end you can check for the value of found to see if the given letter was part of the string. The problem is that the program keeps repeating the loop and checking for the given character even after it found an occurrence of it (which would be an issue with a very long string).

A *classic* alternative would be to turn this into a while loop and check for both conditions (the loop counter and the value of Found):

```
var
  S: string;
  I: Integer;
  Found: Boolean;
begin
  S := 'Hello world';
  Found := False;
  I := Low (S);
  while not Found and (I <= High(S)) do
    begin
      if (S[I]) = 'o' then
        Found := True;
      Inc (I);
    end;
```

While this code is logical and readable, there is more code to write, and if the conditions become multiple and more complex, combining all of the various options would make the code very complicated.

That's why the language (or, to be more precise, its runtime support) has system procedures that let you alter the standard flow of a loop's execution:

- The `Break` procedure interrupts a loop, jumping directly to the first statement following it, skipping any further execution
- The `Continue` procedure jumps to the loop test or counter increment, continuing with the next iteration of the loop (unless the condition is no longer true or the counter has reached its highest value)

Using the `Break` operation, we can modify the original loop for matching a character as follows:

```
var
  S: string;
  I: Integer;
  Found: Boolean;
begin
  S := 'Hello world';
  Found := False;
  for I := Low (S) to High (S) do
    if (S[I]) = 'o' then
      begin
        Found := True;
        Break; // jumps out of the for loop
      end;
end;
```

Two more system procedures, `Exit` and `Halt`, let you immediately return from the current function or procedure or terminate the program. I'll cover `Exit` in the next chapter, while there is basically no reason to ever call `Halt` (so I won't really discuss it in the book).

Here Comes Goto? No Way!

There is actually more to breaking the flow than the four system procedures above. The original Pascal language counted among its features the *infamous* `goto` statement, letting you attach a label to any line of the source code, and jump to that line from another location. Differently from conditional and looping statements, which reveal why you want to diverge from a sequential code flow, `goto` statements generally look like erratic jumps, and are really completely discouraged. Did I mention they are not supported in Object Pascal? No, I didn't, nor am I going to show you a code example. To me `goto` is long gone.

note There are other language statements I haven't covered so far but are part of the language definition. One of them is the `with` statement, which is specifically tied to records, so I'll cover it in Chapter 5. With is another “debated” language feature, but not hated as much as `goto`.

104 - 03: Language Statements

04: procedures and functions

Another important idea emphasized in the Object Pascal language (along with similar features of the C language) is the concept of the routine, basically a series of statements with a unique name, which can be activated many times. Routines (or functions) are called by their name. This way you avoid having to write the same code over and over, and will have a single version of the code used in many places through the program. From this point of view, you can think of routines as a basic code encapsulation mechanism.

Procedures and Functions

In Object Pascal, a routine can assume two forms: a procedure and a function. In theory, a procedure is an operation you ask the computer to perform, a function is a computation returning a value. This difference is emphasized by the fact that a function has a result, a return value, or a type, while a procedure doesn't. The C lan-

106 - 04: Procedures and Functions

guage syntax provides for a single mechanism, functions, and in C a procedure is a function with a `void` (or `null`) result.

Both types of routines can have multiple parameters of specified data types. As we'll see later, procedures and functions are also the basis of the methods of a class, and also in this case the distinction between the two forms remains. In fact, differently from C, C++, Java, C#, or JavaScript, you need to use one of these two keywords when declaring a function or a method.

In practice, even if there are two separate keywords, the difference between functions and procedures is very limited: you can call a function to perform some work and then ignore the result (which might be an optional error code or something like that) or you can call a procedure which passes back a result in one of the parameters (more on reference parameters later in this chapter).

Here is the definition of a procedure using the Object Pascal language syntax, which uses the specific `procedure` keyword and is part of the `FunctionTest` project:

```
procedure Hello;  
begin  
    Show ( 'Hello world! ' );  
end;
```

As a comparison, this would be the same function written with the C language syntax, which has no keyword, requires the parenthesis even in case there are no parameters, and has a `void` or empty return value to indicate no result:

```
void Hello ()  
{  
    Show ("Hello world!");  
};
```

In fact, in the C language syntax there is no difference between procedure and function. In the Pascal language syntax, instead, a function has a specific keyword and must have a return value (or return type).

note There is another very specific syntactical difference between Object Pascal and other languages, that is the presence of a semicolon at the end of the function or procedure signature in the definition, before the `begin` keyword.

There are two ways to indicate the result of the function call, assign the value to function name or use the `Result` keyword:

```
// classic Pascal style  
function DoubleOld (Value: Integer) : Integer;  
begin  
    DoubleOld := Value * 2;  
end;
```

```
// modern alternative
function Double (Value: Integer) : Integer;
begin
    Result := Value * 2;
end;
```

note Differently from the classic Pascal language syntax, modern Object Pascal has actually three ways to indicate the result of a function, including the `Exit` mechanism discussed in this chapter in the section “Exit with a Result”.

The use of `Result` instead of the function name to assign the return value of a function is the most common syntax and tends to make the code more readable. The use of the function name is a classic Pascal notation, now rarely used but still supported.

Again, by comparison the same function could be written with the C language syntax as the following:

```
int Double (int value)
{
    return value * 2;
};
```

note A return statement in languages based on the C syntax indicates the result of the function but also terminates the execution, returning control to the caller. In Object Pascal, instead, assigning a value to the result of a function does not terminate it. That's why the result is often used as a regular variable, for example assigning an initial default or even to modify its result in an algorithm. At the same time, if you need to stop execution you also need to use `Exit` of some other flow control statement. All of this is covered in more details in the following section “Exit with a Result”.

If this is how these routines can be defined, the calling syntax is relatively straightforward, as you type in the identifier followed by the parameters within parenthesis. In cases where there are no parameters, the empty parenthesis can be omitted (unlike languages based on the C syntax). This code snippet and several following ones are part of the `FunctionsTest` project of this chapter:

```
// call the procedure
Hello;

// call the function
X := Double (100);
Y := Double (X);
Show (Y.ToString);
```

This is the encapsulation of code concept that I've introduced. When you call the `Double` function, you don't need to know the algorithm used to implement it. If you later find out a better way to double numbers, you can easily change the code of the function, but the calling code will remain unchanged (although executing it might become faster).

108 - 04: Procedures and Functions

The same principle can be applied to the `Hello` procedure: We can modify the program output by changing the code of this procedure, and the main program code will automatically change its effect. Here is how we can change the procedure implementation code:

```
procedure Hello;  
begin  
    Show ('Hello world, again!');  
end;
```

Forward Declarations

When you need to use an identifier (of any kind), the compiler must have already seen it, to know what the identifier refers to. For this reason, you usually provide a full definition before using any routine. However, there are cases in which this is not possible. If procedure A calls procedure B, and procedure B calls procedure A, when you start writing the code, you will need to call a routine for which the compiler still hasn't seen a definition.

In this cases (and in many others) you can declare the existence of a procedure or function with a certain name and given parameters, without providing its actual code. One way to declare a procedure or functions without defining it is to write its name and parameters (referred to as the function signature) followed by the `forward` keyword:

```
procedure NewHello; forward;
```

Later on, the code should provide a full definition of the procedure (which must be in the same unit), but the procedure can now be called before it is fully defined. Here is an example, just to give you the idea:

```
procedure DoubleHello; forward;  
  
procedure NewHello;  
begin  
    if MessageDlg ('Do you want a double message?',  
        TMsgDlgType.mtConfirmation,  
        [TMsgDlgBtn.mbYes, TMsgDlgBtn.mbNo],  
        0) = mrYes then  
        DoubleHello  
    else  
        ShowMessage ('Hello');  
end;  
  
procedure DoubleHello;  
begin  
    NewHello;  
    NewHello;
```

| end;

note The `MessageDlg` function called in the previous snippet is a relatively simple way to ask a confirmation of the user in the FireMonkey framework (a similar functions exists in the VCL framework as well). The parameters are the message, the type of dialog box, and buttons you want to display. The result is the identifier of the button that was selected.

This approach (which is also part of the `FunctionTest` application project) allows you to write mutual recursion: `DoubleHello` calls `Hello`, but `Hello` might call `DoubleHello` too. In other words, if you keep selecting the Yes button the program will continue showing the message, and show each twice for every Yes. In recursive code, there must be a condition to terminate the recursion, to avoid a condition known as stack overflow.

note Function calls use the *stack* portion of the application memory for the parameters, the return value, local variables and more. If a functions keeps calling itself in an endless loop, the memory area for the stack (which is generally of a fixed and predefined size, determined by the linker and configured in the project options) will terminate through an error known as a *stack overflow*. Needless to say that the popular developers support site (www.stackoverflow.com) took its name from this programming error.

Although a forward procedure declaration is not very common in Object Pascal, there is a similar case that is much more frequent. When you declare a procedure or function in the interface section of a unit, it is automatically considered as a forward declaration, even if the `forward` keyword is not present. Actually you cannot write the body of a routine in the interface section of a unit. Notice that you must provide the actual implementation of each routine you have declared in the same unit.

A Recursive Function

Given I mentioned recursion and gave a rather peculiar example of it (with two procedures calling each other), let me also show you a classic example of a recursive function calling itself. Using recursion is often an alternative way to code a loop.

To stick with a classic demo, suppose you want to compute the power of a number, and you lack the proper function (which is available in the run-time library, of course). You might remember from math, that 2 at the power of 3 corresponds to multiplying 2 by itself 3 times, that is $2*2*2$.

One way to express this in code would be to write a `for` loop that is executed 3 times (or the value of the exponent) and multiplies 2 (or the value of the base) by the current total, starting with 1:

110 - 04: Procedures and Functions

```
function PowerL (Base, Exp: Integer): Integer;  
var  
    I: Integer;  
begin  
    Result := 1;  
    for I := 1 to Exp do  
        Result := Result * Base;  
end;
```

An alternative approach is to repeatedly multiply the base by the power of the same number, with a decreasing exponent, until the exponent is 0, in which case the result is invariably 1. This can be expressed by calling the same function over and over, in a recursive way:

```
function PowerR (Base, Exp: Integer): Integer;  
var  
    I: Integer;  
begin  
    if Exp = 0 then  
        Result := 1  
    else  
        Result := Base * PowerR (Base, Exp - 1);  
end;
```

The recursive version of the program is likely not faster than the version based on the `for` loop, nor more readable. However there are scenarios such as parsing code structures (a tree structure for example) in which there isn't a fixed number of elements to process, and hence writing a loop is close to impossible, while a recursive functions adapts itself to the role.

In general, though, recursive code is powerful but tends to be more complex. After many years in which recursion was almost forgotten, compared to the early days of programming, new functional languages such Haskell, Erlang and Elixir make heavy use of recursion and are driving this idea back to popularity.

In any case, you can find the two power functions in the code in the `FunctionTest` application project.

note The two power functions of the demo don't handle the case of a negative exponent. The recursive version in such a case will loop forever (an until the program hits a physical constraint). Also, by using integers it is relatively fast to reach the maximum data type size and overflow it. I wrote these functions with these inherent limitations to try to keep their code simple.

What Is a Method?

We have seen how you can write a forward declaration in the interface section of a unit of by using the `forward` keyword. The declaration of a method inside a class type is also considered a forward declaration.

But what exactly is a method? A method is a special kind of function or procedure that is related to one of two data types, a record or a class. In Object Pascal, every time we handle an event for a visual component, we need to define a method, generally a procedure, but the term method is used to indicate both functions and procedures tied to a class or record.

Here is an empty method automatically added to the source code of a form (which is indeed a class, as we'll explore much later in the book):

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    // here goes your code
end;
```

Parameters and Return Values

When you call a function or procedure you need to pass the correct number of parameters and make sure they match the expected type. If not, the compiler will issue an error message, similar to a type mismatch when you assign to a variable a value of a wrong type. Given the previous definition of the `Double` function, taking an `Integer` parameter, if you call:

```
Double (10.0);
```

The compiler will show the error:

```
[dcc32 Error] E2010 Incompatible types: 'Integer' and 'Extended'
```

tip

The editor helps you by suggesting the parameters list of a function or procedure with a fly-by hint as soon as you type its name and the open parenthesis. This feature is called Code Parameters and is part of the Code Insight technology (known in other IDEs as *IntelliSense*). CodeInsight starting with Delphi 10.4 is powered by an LSP Server (Language Server Protocol).

There are scenarios in which limited type conversion is allowed, similarly to assignments, but in general you should try to use parameters of the specific type (this is compulsory for reference parameters, as we'll see in a while).

112 - 04: Procedures and Functions

When you call a function, you can pass an expression as a parameter instead of a value. The expression is evaluated and its result assigned to the parameter. In simpler cases, you just pass the name of a variable. In this case, the value of the variable is copied to the parameter (which generally has a different name). I strongly discourage you to use the same name for a parameter and for a variable passed as the value of that parameter, because this can be quite confusing.

warning With Delphi, you should generally not rely on the order of evaluation of the parameters passed to a function, as this varies depending on the calling convention and in some cases it is undefined, although the most common case is right-to-left evaluation. More information at: [http://docwiki.embarcadero.com/RADStudio/en/Procedures_and_Functions_\(Delphi\)#Calling_Conventions](http://docwiki.embarcadero.com/RADStudio/en/Procedures_and_Functions_(Delphi)#Calling_Conventions)

Finally, notice that you can have a function or procedure with different versions (a feature called *overloading*) and with parameters you can skip to let them use a pre-defined value (a feature called *default parameters*). These two key features for functions and procedures are detailed in specific sections later in this chapter.

Exit with a Result

We have seen that returning a result from a function uses quite a different syntax compared to the C family of languages. Not only the syntax is different, but also the behavior. Assigning a value to `Result` (or to the function name) doesn't terminate the function as a `return` statement does. Object Pascal developers often take advantage of this feature, by using `Result` as a temporary storage. Rather than writing:

```
function ComputeValue: Integer;
var
  Value: Integer;
begin
  Value := 0;
  while ...
    Inc (Value);
  Result := Value;
end;
```

You can omit the temporary variable and use `Result` instead. Whatever value `Result` has when the function terminates, is the value returned by the function:

```
function ComputeValue: Integer;
begin
  Result := 0;
  while ...
    Inc (Result);
end;
```

On the other hand there are situations in which you want to assign a value and exit from the procedure right away, for example in a specific `if` branch. If you need to assign the function result *and* stop the current execution you have to use two separate statements, assign the `Result` and then use the `Exit` keyword.

If you remember the code of the `FlowTest` application project of the last chapter (covered in the section “Breaking the Flow with Break and Continue”), this could be rewritten as a function, replacing the call to `Break` with a call to `Exit`. I’ve made this change in the following code snippet, part of the `ParamsTest` application project:

```
function CharInString (S: string; Ch: Char): Boolean;
var
  I: Integer;
begin
  Result := False;
  for I := Low (S) to High (S) do
    if (S[I]) = Ch then
      begin
        Result := True;
        Exit;
      end;
  end;
```

In Object Pascal you can replace the two statements of the `if` block with a special call to `Exit` passing to it the return value of the function, in a way resembling the C language `return` statement. So you can write the code above in a more compact way (also because with a single statement you can avoid the `begin-end` block):

```
function CharInString2 (S: string; Ch: Char): Boolean;
var
  I: Integer;
begin
  Result := False;
  for I := Low (S) to High (S) do
    if (S[I]) = Ch then
      Exit (True);
  end;
```

note `Exit` in Object Pascal is a function so you must enclose the value to be returned in parentheses whereas `return` in C-style languages `return` is a compiler keyword not requiring parentheses.

Reference Parameters

In Object Pascal, procedures and functions allow parameter passing by value and by reference. Passing parameters by value is the default: the value is copied on the stack and the routine uses and manipulates this copy of the data, not the original value (as I described earlier in the section “Function Parameters and Return Val-

114 - 04: Procedures and Functions

ues”). Passing a parameter by reference means that its value is not copied onto the stack in the formal parameter of the routine. Instead, the program refers to the original value, also in the code of the routine. This allows the procedure or function to change the actual value of the variable that was passed as parameter. Parameter passing by reference is expressed by the `var` keyword.

This technique is also available in most programming languages, because avoiding a copy often means that the program executes faster. It isn't present in C (where you can just use a pointer), but it was introduced in C++ and other languages based on the C syntax, where you use the `&` (pass by reference) symbol. Here is an example of passing a parameter by reference using the `var` keyword:

```
procedure DoubleTheValue (var value: Integer);  
begin  
    Value := value * 2;  
end;
```

In this case, the parameter is used both to pass a value to the procedure and to return a new value to the calling code. When you write:

```
var  
    X: Integer;  
begin  
    X := 10;  
    DoubleTheValue (X);  
    Show (X.ToString);
```

the value of the X variable becomes 20, because the function uses a reference to the original memory location of X, affecting its original value.

Compared to general parameters passing rules, passing values to reference parameters is subject to more restrictive rules, given what you are passing is not a value, but an actual variable. You cannot pass a constant value as a reference parameter, an expression, the result of a function, or a property. Another rule is you cannot pass a variable of a slightly different type (requiring automatic conversion). The type of the variable and the parameter must match exactly, or as the compiler error message says:

```
[dcc32 Error] E2033 Types of actual and formal var parameters must be identical
```

This is the error message you'll get if you write, for example (this is also part of the ParamsTest application project, but commented out):

```
var  
    C: Cardinal;  
begin  
    C := 10;  
    DoubleTheValue (C);
```

Passing parameters by reference makes sense for ordinal types and for records (as we'll see in the next chapter). These types are often called *value types* because they have by default a pass-by-value and assign-by-value semantic.

Object Pascal objects and strings have a slightly different behavior we'll investigate in more detail later on. Object variables are references, so you can modify the actual data of an object passed as parameter. These types are part of the different group, often indicated as *reference types*.

Beside standard and reference (`var`) parameter types, Object Pascal has also a very unusual kind of parameter specifier, `out`. An `out` parameter has no initial value and it's used only to return a value. Except for not having an initial value, `out` parameters behave like `var` parameters.

note The `out` parameters were introduced for supporting the corresponding concept in Windows' Component Object model (or COM). They can be used to express the developer intent of expecting uninitialized values.

Constant Parameters

As an alternative to reference parameters, you can use a `const` parameter. Since you cannot assign a new value to a constant parameter inside the routine, the compiler can optimize parameter passing. The compiler can choose an approach similar to reference parameters (or a `const` reference in C++ terms), but the behavior will remain similar to value parameters, because the original value cannot be modified by the function.

In fact, if you try to compile the following code (available, but commented out in the `ParamsTest` project), the system will issue an error:

```
function DoubleThevalue (const value: Integer): Integer;
begin
    value := value * 2;      // compiler error
    Result := value;
end;
```

The error message you'll see might not be immediately intuitive, as it says:

```
[dcc32 Error] E2064 Left side cannot be assigned to
```

Constant parameters are quite common for strings, because in this case the compiler can disable the reference counting mechanism obtaining a slight optimization. This is the most common reason for using constant parameters, a feature that makes limited sense for ordinal and scalar types.

note There is a further little known alternative to passing a `const` parameter, which is adding a `ref` attribute to it, like in “`const [ref]`”. This attribute forces the compiler to pass the constant parameter by reference, where by default the compiler will choose to pass a constant parameter by value or by reference depending on the size of the parameter, with result varying depending on the target CPU and platform.

Function Overloading

At times you might want to have two very similar functions with different parameters and a different implementation. While traditionally you'd have to come up with a slight different name for each, modern programming languages let you *overload* a symbol with multiple definitions.

The idea of overloading is simple: The compiler allows you to define two or more functions or procedures using the same name, provided that the parameters are different. By checking the parameters, in fact, the compiler can determine which of the version of the function you want to call.

Consider this series of functions extracted from the `System.Math` unit of the run-time library:

```
function Min (A,B: Integer): Integer; overload;  
function Min (A,B: Int64): Int64; overload;  
function Min (A,B: Single): Single; overload;  
function Min (A,B: Double): Double; overload;  
function Min (A,B: Extended): Extended; overload;
```

When you call `Min (10, 20)`, the compiler determines that you're calling the first function of the group, so the return value will also be an `Integer`.

There are two basic rules of overloading:

- Each version of an overloaded function (or procedure) must be followed by the `overload` keyword (including the first one).
- Among overloaded functions, there must be a difference in the number or in the type of the parameters. Parameter names are not considered, because they are not indicated during the call. Also, the return type cannot be used to distinguish among two overloaded functions.

note There is an exception to the rule you cannot distinguish functions on the return values and it is for the `Implicit` and `Explicit` conversion operators, covered in Chapter 5.

Here are three overloaded versions of a `ShowMsg` procedure I've added to the `overloadTest` example (an application demonstrating both overloading and default parameters):

```
procedure ShowMsg (Str: string); overload;
begin
    Show ('Message: ' + Str);
end;

procedure ShowMsg (FormatStr: string;
    Params: array of const); overload;
begin
    Show ('Message: ' + Format (FormatStr, Params));
end;

procedure ShowMsg (I: Integer; Str: string); overload;
begin
    Show (I.ToString + ' ' + Str);
end;
```

The three functions show a message box with a string, after optionally formatting the string in different ways. Here are the three calls of the program:

```
ShowMsg ('Hello');
ShowMsg ('Total = %d.', [100]);
ShowMsg (10, 'MBytes');
```

And this is their effect:

```
Message: Hello
Message: Total = 100.
Message: 10 MBytes
```

tip The Code Parameters technology of the IDE works very nicely with overloaded procedures and functions. As you type the open parenthesis after the routine name, all the available alternatives are listed. As you enter the parameters, the Code Insight technology uses their type to determine which of the alternatives are still available.

What if you try to call the function with parameters that don't match any of the available overloaded versions? You'll get an error message, of course. Suppose you want to call:

```
ShowMsg (10.0, 'Hello');
```

The error you'll see in this case is a very specific one:

```
[dcc32 Error] E2250 There is no overloaded version of 'ShowMsg' that
can be called with these arguments
```

The fact that each version of an overloaded routine must be properly marked implies that you cannot overload an existing routine of the same unit that is not marked with the `overload` keyword.

118 - 04: Procedures and Functions

The error message you get when you try is:

```
Previous declaration of '<name>' was not marked with the 'overload' directive.
```

You can, however, create a routine with the same name of one that was declared in a *different unit*, given that units act as namespaces. In this case, you are not overloading a function with a new version, but you are replacing the function with a new version, hiding the original one (which can be referenced using the unit name prefix). This is why the compiler won't be able to pick a version based on the parameters, but it will try to match the only version it sees, issuing an error if the parameters types don't match.

Overloading and Ambiguous Calls

When you call an overloaded function, the compiler will generally find a match and work correctly or issue an error if none of the overloaded versions has the right parameters (as we have just seen).

But there is also a third scenario: Given the compiler can do some type conversions for the parameters of a function, there might be different possible conversions for a single call. When the compiler finds multiple versions of a function it can call, and there isn't one that is a perfect type match (which would be picked) it issues an error message indicating that the function call is *ambiguous*.

This is not a common scenario, and I had to build a rather illogical example to show it to you, but it is worth considering the case (as it does happen occasionally in real world).

Suppose you decide to implement two overloaded functions to add integers and floating point numbers:

```
function Add (N: Integer; S: Single): Single; overload;
begin
    Result := N + S;
end;

function Add (S: Single; N: Integer): Single; overload;
begin
    Result := N + S;
end;
```

These functions are in the `OverloadTest` application project. Now you can call them passing the two parameters in any order:

```
Show (Add (10, 10.0).ToString);
Show (Add (10.0, 10).ToString);
```

However the fact is that, in general, a function can accept a parameter of a different type when there is a conversion, like accepting an integer when the function expects a parameter of a floating point type. So what happens if you call:

```
|| Show (Add (10, 10).ToString);
```

The compiler can call the first version of the overloaded function, but it can also call the second version. Not knowing what you are asking for (and know knowing if calling one function or the other will produce the same effect), it will issue an error:

```
|| [dcc32 Error] E2251 Ambiguous overloaded call to 'Add'
   Related method: function Add(Integer; Single): Single;
   Related method: function Add(Single; Integer): Single;
```

tip In the errors pane of the IDE you'll see an error message with the first line above, and a plus sign on the side you can expand to see the following two lines with the details of which overloaded functions the compiler is considering ambiguous.

If this is a real world scenario, and you need to make the call, you can add a manual type conversions call to solve the problem and indicate to the compiler which of the versions of the function you want to call:

```
|| Show (Add (10, 10.ToSingle).ToString);
```

A particular case of ambiguous calls can happen if you use variants, a rather peculiar data type I'll cover only later in the book.

Default Parameters

Another feature related to overloading, is the possibility of providing a default value to some of the parameters of a function, so that you can call the function with or without those parameters. If the parameter is missing in the call, it will take the default value.

Let me show an example (still part of the `OverloadTest` application project). We can define the following encapsulation of the `Show` call, providing two default parameters:

```
|| procedure NewMessage (Msg: string; Caption: string = 'Message';
   Separator: string = ': ');
   begin
     Show (Caption + Separator + Msg);
   end;
```

With this definition, we can call the procedure in each of the following ways:

```
|| NewMessage ('Something wrong here!');
|| NewMessage ('Something wrong here!', 'Attention');
```


120 - 04: Procedures and Functions

```
| NewMessage ( 'Hello', 'Message', '-- ');
```

This is the output:

```
| Message: Something wrong here!  
| Attention: Something wrong here!  
| Message--Hello
```

Notice that the compiler doesn't generate any special code to support default parameters; nor does it create multiple (overloaded) copies of the functions or procedure. The missing parameters are simply added by the compiler to the calling code. There is one important restriction affecting the use of default parameters: You cannot "skip" parameters. For example, you can't pass the third parameter to the function after omitting the second one.

There are a few other rules for the definition and the calls of functions and procedures (and methods) with default parameters:

- In a call, you can only omit parameters starting from the last one. In other words, if you omit a parameter you must also omit the following ones.
- In a definition, parameters with default values must be at the end of the parameters list.
- Default values must be constants. Obviously, this limits the types you can use with default parameters. For example, a dynamic array or an interface type cannot have a default parameter other than `nil`; records cannot be used at all.
- Parameters with defaults must be passed by value or as `const`. A reference (`var`) parameter cannot have a default value.

Using default parameters and overloading at the same time makes it more likely to get you in a situation which confuses the compiler, raising an *ambiguous call* error, as mentioned in the previous section. For example, if I add the following new version of the `NewMessage` procedure to the previous example:

```
| procedure NewMessage (Str: string; I: Integer = 0); overload;  
| begin  
|   Show (Str + ': ' + IntToStr (I))  
| end;
```

then the compiler won't complain, as this is a legitimate definition. However, if you write the call:

```
| NewMessage ( 'Hello' );
```

this is flagged by the compiler as:

```
| [dcc32 Error] E2251 Ambiguous overloaded call to 'NewMessage'  
|   Related method: procedure NewMessage(string; string; string);  
|   Related method: procedure NewMessage(string; Integer);
```

Notice that this error shows up in a line of code that compiled correctly before the new overloaded definition. In practice, we have no way to call the `NewMessage` procedure with one string parameter, as the compiler doesn't know whether we want to call the version with only the string parameter or the one with the string parameter and the integer parameter with a default value. When it has a similar doubt, the compiler stops and asks the programmer to state his or her intentions more clearly.

Inlining

Inlining Object Pascal functions and methods is a low-level language feature that can lead to significant optimizations. Generally, when you call a method, the compiler generates some code to let your program jump to a new execution point. This implies setting up a stack frame and doing a few more operations and might require a dozen or so machine instructions. However, the method you execute might be very short, possibly even an access method that simply sets or returns some private field. In such a case, it makes a lot of sense to copy the actual code to the call location, avoiding the stack frame setup and everything else. By removing this overhead, your program will run faster, particularly when the call takes place in a tight loop executed thousands of times.

For some very small functions, the resulting code might even be smaller, as the code pasted in place might be smaller than the code required for the function call. However, notice that if a longer function is inlined and this function is called in many different places in your program, you might experience code bloat, which is an unnecessary increase in the size of the executable file.

In Object Pascal you can ask the compiler to inline a function (or a method) with the `inline` directive, placed after the function (or method) declaration. It is not necessary to repeat this directive in the definition. Always keep in mind that the `inline` directive is only a hint to the compiler, which can decide that the function is not a good candidate for inlining and skip your request (without warning you in any way). The compiler might also inline some, but not necessarily all, of the calls of the function after analyzing the calling code and depending on the status of the `$INLINE` directive at the calling location. This directive can assume three different values (notice that this feature is independent from the optimization compiler switch):

- With `{ $INLINE OFF }` you can suppress inlining in a program, in a portion of a program, or for a specific call site, regardless of the presence of the `inline` directive in the functions being called.

122 - 04: Procedures and Functions

- With default value, `{$INLINE ON}`, inlining is enabled for functions marked by the `inline` directive.
- With `{$INLINE AUTO}` the compiler will generally inline the functions you mark with the directive, plus automatically inline very short functions. Watch out because this directive can cause code bloat.

There are many functions in the Object Pascal Run-Time Library that have been marked as inline candidates. For example, the `Max` function of the `System.Math` unit has definitions like:

```
function Max(const A, B: Integer): Integer;  
  overload; inline;
```

To test the actual effect of inlining this function, I've written the following loop in the `InliningTest` application project:

```
var  
  Sw: TStopwatch;  
  I, J: Integer;  
begin  
  J := 0;  
  Sw := TStopwatch.StartNew;  
  for I := 0 to LoopCount do  
    J := Max (I, J);  
  Sw.Stop;  
  Show ('Max ' + J.ToString +  
    ' [' + Sw.ElapsedMilliseconds.ToString + ']');
```

In this code, the `TStopwatch` record of the `System.Diagnostics` unit, a structure that keep track of the time (or system ticks) elapsed between the start (or `StartNew`) and the `Stop` calls.

The form has two buttons both calling this same exact code, but one of them has inlining disabled at the call site. Notice you need to compile with the Release configuration to see any difference (as inlining is a Release optimization). With twenty million interactions (the value of the `LoopCount` constant), on my computer I get the following results:

```
// on windows (running in a VM)  
Max on 20000000 [17]  
Max off 20000000 [45]  
  
// on Android (on device)  
Max on 20000000 [280]  
Max off 20000000 [376]
```

How can we read this data? On Windows, inlining more than doubles the execution speed, while on Android it makes the program about 35% faster. However, on a device the program runs much slower (an order of magnitude) so while on Windows

we shave off 30 milliseconds on my Android device this optimization saves about 100 milliseconds.

The same program does a second similar test with the `Length` function, a compiler-magic function that was specifically modified to be inlined. Again the inlined version is significantly faster on both Windows and Android:

```
// on windows (running in a VM)
Length inlined 260000013 [11]
Length not inlined 260000013 [40]

// on Android (on device)
Length inlined 260000013 [401]
Length not inlined 260000013 [474]
```

This is the code used by this second testing loop:

```
var
  Sw: TStopwatch;
  I, J: Integer;
  Sample: string;
begin
  J := 0;
  Sample:= 'sample string';
  Sw := TStopwatch.StartNew;
  for I := 0 to LoopCount do
    Inc (J, Length(Sample));
  Sw.Stop;
  Show ('Length not inlined ' + IntToStr (J) +
    ' [' + IntToStr (Sw.ElapsedMilliseconds) + ' ] ');
end;
```

The Object Pascal compiler doesn't define a clear cut limit on the size of a function that can be inlined or a specific list of constructs (`for` or `while` loops, conditional statements) that would prevent inlining. However, since inlining a large function provides little advantage yet exposes you to the risk of some real disadvantages (in terms of code bloat), you should avoid it.

One limitation is that the method or function cannot reference identifiers (such as types, global variables, or functions) defined in the implementation section of the unit, as they won't be accessible in the call location. However, if you are calling a local function, which happens to be inlined as well, then the compiler will accept your request to inline your routine.

A drawback is that inlining requires more frequent recompilations of units, as when you modify an inlined function, the code of each of the calling sites will need to be recompiled as well. Within a unit, you might write the code of the inlined functions before calling them, but better place them at the beginning of the implementation section.

note Delphi has a single pass compiler, so it cannot refer to the code of a function it hasn't seen yet.

Within different units, you need to specifically add other units with inlined functions to your `uses` statements, even if you don't call those methods directly. Suppose your unit A calls an inlined function defined in unit B. If this function in turn calls another inlined function in unit C, your unit A needs to refer to C as well. If not, you'll see a compiler warning indicating the call was not inlined due to the missing unit reference. A related effect is that functions are never inlined when there are circular unit references (through their implementation sections).

Advanced Features of Functions

If what I have covered so far includes the core features related to functions, there are several advanced capabilities worth exploring. If you are really a newbie in terms of software development, however, you might want to skip the rest of this chapter for now and move to the next one.

Object Pascal Calling Conventions

Whenever your code calls a function, the two sides need to agree on the actual practical way parameters are passed from the caller to the callee, something called *calling convention*. Generally, a function call takes place by passing the parameters (and expecting the return value) via the stack memory area. However, the order in which the parameters and return value are placed on the stack changes depending on the programming language and platform, with most languages capable of using multiple different calling conventions.

A long time ago, the 32-bit version of Delphi introduced a new approach to passing parameters, known as "*fastcall*": Whenever possible, up to three parameters can be passed in CPU registers, making the function call much faster. Object Pascal uses this fast calling convention by default although it can also be requested by using the `register` keyword.

Fastcall is the default calling convention, and functions using it are not compatible with external libraries, like Windows API functions in Win32. The functions of the Win32 API must be declared using the `stdcall` (*standard call*) calling convention, a mixture of the original `pascal` calling convention of the Win16 API and the `cdecl`

calling convention of the C language. All of these calling conventions are supported in Object Pascal, but you'll rarely use something different than the default unless you need to invoke a library written in a different language, like a system library.

The typical case you need to move away from the default fast calling convention is when you need to call the native API of a platform, which requires a different calling convention depending on the operating system. Even Win64 uses a different model to Win32, so Object Pascal supports many different options, not really worth detailing here. Mobile operating systems, instead, tend to expose classes, rather than native functions, although the issue of respecting a given calling convention has to be taken into account even in that scenario.

Procedural Types

Another feature of Object Pascal is the presence of procedural types. These are really an advanced language topic, which only a few programmers will use regularly. However, since we will discuss related topics in later chapters (specifically, method pointers, a technique heavily used by the environment to define event handlers, and anonymous methods), it's worth giving a quick look at them here.

In Object Pascal (but not in the more traditional Pascal language) there is the concept of a procedural type (which is similar to the C language concept of a function pointer – something languages like C# and Java have dropped, because it is tied to global functions and pointers).

The declaration of a procedural type indicates the list of parameters and, in the case of a function, the return type. For example, you can declare a new procedural type, with an Integer parameter passed by reference, with this code:

```
type
  TIntProc = procedure (var Num: Integer);
```

This procedural type is compatible with any routine having exactly the same parameters (or the same function signature to use C jargon). Here is an example of a compatible routine:

```
procedure DoubleTheValue (var value: Integer);
begin
  value := value * 2;
end;
```

Procedural types can be used for two different purposes: you can declare variables of a procedural type or pass a procedural type (that is, a function pointer) as parameter to another routine. Given the preceding type and procedure declarations, you can write this code:

126 - 04: Procedures and Functions

```
var
  IP: TIntProc;
  X: Integer;
begin
  IP := DoubleTheValue;
  X := 5;
  IP (X);
end;
```

This code has the same effect as the following shorter version:

```
var
  X: Integer;
begin
  X := 5;
  DoubleTheValue (X);
end;
```

The first version is clearly more complex, so why and when should we use it? There are cases in which being able to decide which function to call and actually calling it later on can be very powerful. It is possible to build a complex example showing this approach. However, I prefer to let you explore a fairly simple application project, called `ProcType`.

This example is based on two procedures. One procedure is used to double the value of the parameter like the one I've already shown. A second procedure is used to triple the value of the parameter, and therefore is named `TripleTheValue`:

```
procedure TripleTheValue (var value: Integer);
begin
  value := value * 3;
end;
```

Instead of calling these functions directly, one or the other are saved in a procedural type variable. The variable is modified as a users selects a checkbox, and the current procedure is called in this generic way as a user clicks the button. The program uses two initialized global variables (the procedure to be called and the current value), so that these values are preserved over time. This is the full code, save for the definitions of the actual procedures, already shown above:

```
var
  IntProc: TIntProc = DoubleTheValue;
  Value: Integer = 1;

procedure TForm1.CheckBox1Change(Sender: TObject);
begin
  if CheckBox1.IsChecked then
    IntProc := TripleTheValue
  else
    IntProc := DoubleTheValue;
end;

procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
  IntProc (Value);
  Show (Value.ToString);
end;
```

When the user changes the check box status, all following button clicks will call the active function. So if you press the button twice, change the selection, and press the button twice again, you'll first double twice and then triple twice the current value, producing the following output:

```
2
4
12
36
```

Another practical example of the use of procedural types is when you need to pass a function to an operating system like Windows (where they are generally called “call-back functions”). As mentioned at the beginning of this section, in addition to procedural types Object Pascal developers use method pointers (covered in Chapter 10) and anonymous methods (covered in Chapter 15).

note The most common object oriented mechanism to obtain a late bound function call (that is a function call that can change at runtime) is the use of virtual methods. While virtual methods are very common in Object Pascal, procedural types are seldom used. The technical foundation, though, is somehow similar. Virtual functions and polymorphism are covered in Chapter 8.

External Functions Declarations

Another important element for system programming is represented by external declarations. Originally used to link code to external functions that were written in assembly language, external declarations became commonplace Windows programming to call a function from a DLL (a dynamic link library). An external function declaration implies the ability to call a function not fully available to the compiler or the linker, but requiring the ability to load an external dynamic library and invoke one of its functions.

note Whenever you call an API for a given platform in your Object Pascal code you lose the ability to recompile the application for any other platform than the specific one. The exception is if the call is surrounded by platform specific `$IFDEF` compiler directives.

This is, for example, how you can invoke Windows API functions from a Delphi application. If you open the `Winapi.Windows` unit you'll find many function declarations and definitions like:

128 - 04: Procedures and Functions

```
// forward declaration
function GetUserName(lpBuffer: LPWSTR;
    var nSize: DWORD): BOOL; stdcall;

// external declaration (instead of actual code)
function GetUserName; external advapi32
    name 'GetUserNamew';
```

You seldom need to write declarations like the one just illustrated, since they are already listed in the `windows` unit and many other system units. The only reason you might need to write this external declaration code is to call functions from a custom DLL, or to call Windows functions not translated in the platform API.

This declaration means that the code of the function `GetUserName` is stored in the `advapi32` dynamic library (`advapi32` is a constant associated with the full name of the DLL, `'advapi32.dll'`) with the name `GetUserNamew`, as this API function has both an ASCII and a WideString version. Inside an external declaration, in fact, we can specify that our function refers to a function of a DLL that originally had a different name.

Delayed Loading of DLL Functions

In the Windows operating system, there are two ways to invoke an API function of the Windows SDK (or any other DLL): you can let the application loader resolve all references to external functions or you can write specific code that looks for a function and executes it if available.

The former code is easier to write (as we saw in the previous section): as all you need is the external function declaration. However if the library or even just one of the functions you want to call is not available, your program will not be able to start on the operating system versions that don't provide that function.

Dynamic loading allows for more flexibility, but implies loading the library manually, using the `GetProcAddress` API for finding the function you want to call, and invoking it after casting the pointer to the proper type. This kind of code is quite cumbersome and error prone.

That's why it is good that the Object Pascal compiler and linker have specific support for a feature now available at the Windows operating system level and already used by some C++ compilers, the delayed loading of functions until the time they are called. The aim of this declaration is not to avoid the implicit loading of the DLL, which takes place anyway, but to allow the delayed binding of that specific function within the DLL.

You basically write the code in a way that's very similar to the classic execution of DLL function, but the function address is resolved the first time the function is

called and not at load time. This means that if the function is not available you get a run-time exception, `EExternalException`. However, you can generally verify the current version of the operating system or the version of the specific library you are calling, and decide in advance whether you want to make the call or not.

note If you want something more specific and easier to handle at a global level than an exception, you can hook into the error mechanism for the delayed loading call, as explained by Allen Bauer in his blog post: https://blog.therealoracleatdelphi.com/2009/08/exceptional-procrastination_29.html

From the Object Pascal language perspective, the only difference is in the declaration of the external function. Rather than writing:

```
function MessageBox;  
external user32 name 'MessageBoxW';
```

You can now write (again, from an actual example in the windows unit):

```
function GetSystemMetricsForDpi(nIndex: Integer; dpi: UINT): Integer;  
stdcall; external user32 name 'GetSystemMetricsForDpi' delayed;
```

At run time, considering that the API was added to Windows 10, version 1607 for the first time, you might want to write code like the following:

```
if (TOSVersion.Major >= 10) and (TOSVersion.Build >= 14393)  
begin  
    NMetric := GetSystemMetricsForDpi (SM_CXBORDER, 96);
```

This is much, much less code than you had to write without delayed loading to be able to run the same program on older versions of Windows.

Another relevant observation is that you can use the same mechanism when building your own DLLs and calling them in Object Pascal, providing a single executable that can bind to multiple versions of the same DLL as long as you use delayed loading for the new functions.

130 - 04: Procedures and Functions

05: arrays and records

When I introduced data types in Chapter 2, I referred to the fact that in Object Pascal there are both built in data types and type constructors. A simple example of a type constructor is the enumerated type, covered in that chapter.

The real power of type definition comes with more advanced mechanisms, such as arrays, records, and classes. In this chapter I'll cover the first two, which in their essence date back to the early definition of Pascal, but have been changed so much over the years (and made so powerful) that they barely resemble their ancestral type constructors with the same name.

Towards the end of the chapter I'll also briefly introduce some advanced Object Pascal data types as pointers. The real power of custom data types, however, will be unveiled in Chapter 7, where we'll start looking into classes and object-oriented programming.

Array Data Types

Array types define lists with elements of a specific type. These lists can have a fixed number of elements (static arrays) or of a variable number of elements (dynamic arrays). You generally use an *index* within square brackets to access one of the elements of an array. Square brackets are also used to specify the number of values of a fixed size array.

The Object Pascal language supports different array types, from traditional static arrays to dynamic ones. Use of dynamic arrays is recommended, particularly with the mobile versions of the compiler. I'll introduce static arrays first, and later focus on dynamic ones.

Static Arrays

Traditional Pascal language arrays are defined with a static or fixed size. An example is in the following code snippets, which defines a list of 24 integers, presenting the temperatures during the 24 hours of a day:

```
type
  TDayTemperatures = array [1..24] of Integer;
```

In this classic array definition, you can use a subrange type within square brackets, actually defining a new specific subrange type using two constants of an ordinal type. This subrange indicates the valid indexes of the array. Since you specify both the upper and the lower index of the array, the indexes don't need to be zero-based, as it is the case in C, C++, Java, and most other languages (although 0-based arrays are also quite common in Object Pascal). Notice also that static array indexes in Object Pascal can be numbers, but also other ordinal types like characters, enumerated types, and more. Non-integral indexes are quite rare, though.

note There are languages like JavaScript that make heavy use of associative arrays. Object Pascal arrays are limited to ordinal indexes, so you cannot directly use a string as index. There are ready to use data structures in the RTL implementing Dictionaries and other similar data structures that provide such features. I'll cover them in the chapter about Generics, in the third part of the book.

Since the array indexes are based on subranges, the compiler can check their range. An invalid constant subrange results in a compile-time error; and an out-of-range index used at run-time results in a run-time error, but only if the corresponding compiler option is enabled.

note This is the *Range checking* option of the *Runtime errors* group of the *Compiling* page of the Project Options dialog of the IDE. I've already mentioned this option in Chapter 2, in the section "Subrange Types".

Using the array definition above, you can set the value of a `DayTemp1` variable of the `TDayTemperatures` type as follows (and as I've done in the `ArraysTest` application project, from which the following code snippets have been extracted):

```
type
  TDayTemperatures = array [1..24] of Integer;

var
  DayTemp1: TDayTemperatures;

begin
  DayTemp1 [1] := 54;
  DayTemp1 [2] := 52;
  ...
  DayTemp1 [24] := 66;

  // The following line causes:
  // E1012 Constant expression violates subrange bounds
  // DayTemp1 [25] := 67;
```

Now a standard way to operate on arrays, given their nature, is to use for cycles. This is an example of a loop used to display all of the temperatures for a day:

```
var
  I: Integer;
begin
  for I := 1 to 24 do
    Show (I.ToString + ': ' + DayTemp1[I].ToString);
```

While this code works, having hard-coded the array boundaries (1 and 24) is far from ideal, as the array definition itself might change over time and you might want to move to using a dynamic array.

Array Size and Boundaries

When you work with an array, you can always test its boundaries by using the standard `Low` and `High` functions, which return the lower and upper bounds. Using `Low` and `High` when operating on an array is highly recommended, especially in loops, since it makes the code independent of the current range of the array (which might go from 0 to the length of the array minus one, might start from 1 and reach the array's length, or have any other subrange definition). If you should later change the declared range of the array indexes, code that uses `Low` and `High` will still work. If you write a loop hard-coding the range of an array you'll have to update the code of

134 - 05: Arrays and Records

the loop when the array size changes. `Low` and `High` make your code easier to maintain and more reliable.

note Incidentally, there is no run-time overhead for using `Low` and `High` with static arrays. They are resolved at compile-time into constant expressions, not actual function calls. This compile-time resolution of expressions and function calls also happens for many other system functions.

Another relevant function is `Length`, which returns the number of elements of the array. I've combined these three functions in the following code that computes and displays the average temperature for the day:

```
var
  I: Integer;
  Total: Integer;
begin
  Total := 0;
  for I := Low(DayTemp1) to High(DayTemp1) do
    Inc (Total, DayTemp1[I]);
  Show ((Total / Length(DayTemp1)).ToString);
```

This code is also part of the `ArraysTest` application project.

Multi-Dimensional Static Arrays

An array can have more than one dimension, expressing a matrix or a cube rather than a list. Here are two sample definitions:

```
type
  TAllMonthTemps = array [1..24, 1..31] of Integer;
  TAllYearTemps = array [1..24, 1..31, 1..12] of Integer;
```

You can access an element as:

```
var
  AllMonth1: TAllMonthTemps;
  AllYear1: TAllYearTemps;
begin
  AllMonth1 [13, 30] := 55; // hour, day
  AllYear1 [13, 30, 8] := 55; // hour, day, month
```

note Static arrays immediately take up a lot of memory (in the case above on the stack), which should be avoided. The `AllYear1` variable requires 8,928 Integers, taking up 4 bytes each, that is almost 35KB. Allocating such a large block in the global memory or on the stack (as in the demo code) is really a mistake. A dynamic array, instead, uses the heap memory, and offers much more flexibility in terms of memory allocation and management.

Given these two array types are built on the same core types, you should better declare them using the preceding data types, as in the following code:

```
type
  TMonthTemps = array [1..31] of TDayTemperatures;
  TYearTemps = array [1..12] of TMonthTemps;
```

This declaration inverts the order of the indexes as presented above, but it also allows assignment of whole blocks between variables. Let's see how you can assign individual values:

```
Month1 [30][14] := 44;
Month1 [30, 13] := 55; // day, hour
Year1 [8, 30, 13] := 55; // month, day, hour
```

In theory, you should use the first line, selecting one of the array of arrays, and then an element of the resulting array. However, the version with the two indexes within square brackets is also allowed. Or with three indexes, in the “cube” example.

The importance of using intermediate types lies on the fact that arrays are type compatible only if they refer to the same exact type name (that is exactly the same type definition) not if their type definitions happen to refer to the same implementation. This type compatibility rule is the same for all types in the Object Pascal, with only some specific exceptions.

For example, the following statement copies a months's temperatures to the third month of the year:

```
Year1[3] := Month1;
```

Instead, a similar statement based on the stand alone array definitions (which are not type compatible):

```
AllYear1[3] := AllMonth1;
```

would cause the error:

```
Error: Incompatible types: 'array[1..31] of array[1..12] of Integer'
and 'TAllMonthTemps'
```

As I mentioned, static arrays suffer memory management issues, specifically when you want to pass them as parameters or allocate only a portion of a large array. Moreover, you cannot resize them during the lifetime of the array variable. This is why is preferable to use dynamic arrays, even if they require a little extra management, for example regarding memory allocation.

Dynamic Arrays

In the traditional Pascal language arrays had a fixed-size arrays and you specified the number of elements of the array as you declared the data type. Object Pascal supports also a direct and native implementation of dynamic arrays.

note “Direct implementation of dynamic arrays” here is in contrast to using pointers and dynamic memory allocation to obtain a similar effect... with very complex and error-prone code. By the way, dynamic arrays is the only flavor of the construct in most modern programming languages.

Dynamic arrays are dynamically allocated and reference counted (making parameter passing much faster, as only the reference is passed, and not a copy of the complete array). When you are done, you can clear an array by setting its variable to `nil` or its length to zero, but given they are reference counted in most cases the compiler will automatically free the memory for you.

With a dynamic array, you declare an array type without specifying the number of elements and then allocate it with a given size using the `SetLength` procedure:

```
var
  Array1: array of Integer;
begin
  // this would cause a runtime Range Check error
  // Array1 [1] := 100;
  SetLength (Array1, 10);
  Array1 [1] := 100; // this is OK
```

You cannot use the array until you've assigned its length, allocating the required memory on the heap. If you do so, you'd either see a Range Check error (if the corresponding compiler option is active) or an Access Violation (on Windows) or similar memory access error on another platform. The `SetLength` call sets all the values to zero. The initialization code makes it possible to start reading and writing values of the array right away, without any fear of memory errors (unless you violate the array boundaries).

If you do need to allocate memory explicitly, you don't need to free it directly. In the code snippet above, as the code terminates and the `Array1` variable goes out of scope, the compiler will automatically free its memory (in this case the ten integers that have been allocated). So while you can assign a dynamic array variable to `nil` or call `SetLength` with 0 value, this is generally not needed (and rarely done).

Notice that the `SetLength` procedure can also be used to resize an array, without losing its current content (if you are growing it) but you would lose some elements (if you are shrinking it). As in the initial `SetLength` call you indicate only the number of elements of the array, the index of a dynamic array invariably starts from 0 and goes

up to the number of elements minus 1. In other words, dynamic arrays don't support two features of classic static Pascal arrays, the non-zero low bound and the non-integer indexes. At the same time, they match more closely how arrays work in most languages based on the C syntax.

Just like static arrays, to know about the current size of a dynamic array, you can use the `Length`, `High`, and `Low` functions. For dynamic arrays, however, `Low` always returns 0, and `High` always returns the length minus one. This implies that for an empty array `High` returns -1 (which, when you think about it, is a strange value, as it is lower than that returned by `Low`).

So, as an example, in the `DynArray` application project I've populated and extracted the information from a dynamic array using adaptable loops. This is the type and variable definition:

```
type
  TIntegersArray = array of Integer;
var
  IntArray1: TIntegersArray;
```

The array is allocated and populated with values matching the index, using the following loop:

```
var
  I: Integer;
begin
  SetLength (IntArray1, 20);
  for I := Low (IntArray1) to High (IntArray1) do
    IntArray1 [I] := I;
end;
```

A second button has the code both to display each value and compute the average, similar to that of the previous example but in a single loop:

```
var
  I: Integer;
  Total: Integer;
begin
  Total := 0;
  for I := Low(IntArray1) to High(IntArray1) do
    begin
      Inc (Total, IntArray1[I]);
      Show (I.ToString + ': ' + IntArray1[I].ToString);
    end;
  Show ('Average: ' + (Total / Length(IntArray1)).ToString);
end;
```

The output of this code is quite obvious (and mostly omitted):

```
0: 0
1: 1
2: 2
3: 3
```

138 - 05: Arrays and Records

```
...
17: 17
18: 18
19: 19
Average: 9.5
```

Beside `Length`, `SetLength`, `Low`, and `High`, there are also other common procedures that you can use on arrays, such as the `Copy` function, you can use to copy a portion of an array (or all of it). Notice that you can also assign an array from a variable to another, but in that case you are not making a full copy, but rather having two variables referring to the same array in memory.

The only slightly complex code is in the final part of the `DynArray` application project, which copies one array to the other in two different ways:

- using the `Copy` function, which duplicates the array data in a new data structure using a separate memory area
- using the assignment operator, which effectively creates an alias, a new variable referring to the same array in memory

At this point, if you modify one of the elements of the new arrays, you will affect the original version or not depending on the way you made the copy. This is the complete code:

```
var
  IntArray2: TIntegersArray;
  IntArray3: TIntegersArray;
begin
  // alias
  IntArray2 := IntArray1;

  // separate copy
  IntArray3 := Copy (IntArray1, Low(IntArray1), Length(IntArray1));

  // modify items
  IntArray2 [1] := 100;
  IntArray3 [2] := 100;

  // check values for each array
  Show (Format ('[%d] %d -- %d -- %d',
    [1, IntArray1 [1], IntArray2 [1], IntArray3 [1]]));
  Show (Format ('[%d] %d -- %d -- %d',
    [2, IntArray1 [2], IntArray2 [2], IntArray3 [2]]));
```

The output you'll get is like the following:

```
[1] 100 -- 100 -- 1
[2] 2 -- 2 -- 100
```

The changes to `IntArray2` propagate to `IntArray1`, because they are just two references to the same physical array; the changes to `IntArray3` are separate, because it has a separate copy of the data.

Native Operations on Dynamic Arrays

Dynamic arrays have support for assigning constant arrays and for concatenation.

note These extensions to dynamic arrays were added in Delphi XE7 and help making developer feel this feature valuable and the obvious choice (forgetting about their static counterpart).

In practice, you can write code like the following, which is significantly simplified from earlier code snippets:

```
var
  DI: array of Integer;
  I: Integer;
begin
  DI := [1, 2, 3];    // initialization
  DI := DI + DI;      // concatenation
  DI := DI + [4, 5];  // mixed concatenation

  for I in DI do
  begin
    Show (I.ToString);
  end;
```

Notice the use of a for-in statement to scan the array elements in this code, which is part of the `DynArrayConcat` application project. Notice that these arrays can be based on any data type, from simple integers like in the code above, to record and classes.

There is a second addition that was done alongside with assignment and concatenation, but that is part of the RTL more than the language. It is now possible to use on dynamic arrays functions that were common for strings, like `Insert` and `Delete`.

This means you can now write code like the following (part of the same project):

```
var
  DI: array of Integer;
  I: Integer;
begin
  DI := [1, 2, 3, 4, 5, 6];
  Insert ([8, 9], DI, 4);
  Delete (DI, 2, 1); // remove the third (0-based)
```

Open Array Parameters

There is a very special scenario for the use of arrays, which is passing a flexible list of parameters to a function. Beside passing an array directly, there are two special syntax structures explained in this and the next section. An example of such a func-

140 - 05: Arrays and Records

tion, by the way, is the `Format` function that I called in the last code snippet and that has an array of values in square brackets as its second parameter.

Unlike the C language (and some of the other languages based on C syntax), in the traditional Pascal language a function or procedure always has a fixed number of parameters. However, in Object Pascal there is a way to pass a varying number of parameters to a routine using as parameter an array, a technique known as *open array parameters*.

note Historically, open array parameters predate dynamic arrays, but today these two features look so similar in the way they work that they are almost indistinguishable these days. That's why I covered open array parameters only after discussing dynamic arrays.

The basic definition of an open array parameter is the same as that of a typed dynamic array type, prefixed by the `const` specifier. This means you indicate the type of the parameter(s), but you don't need to indicate how many elements of that type the array is going to have. Here is an example of such a definition, extracted from the `OpenArray` application project:

```
function Sum (const A: array of Integer): Integer;
var
  I: Integer;
begin
  Result := 0;
  for I := Low(A) to High(A) do
    Result := Result + A[I];
end;
```

You can call this function by passing to it an *array-of-Integer* constant expression (which can also include variables as part of the expressions used to compute the individual values):

```
X := Sum ([10, Y, 27*I]);
```

Given a dynamic array of `Integer`, you can pass it directly to a routine requiring an open array parameter of the same base type (Integers in this case). Here is an example, where the complete array is passed as parameter:

```
var
  List: array of Integer;
  X, I: Integer;
begin
  // initialize the array
  SetLength (List, 10);
  for I := Low (List) to High (List) do
    List [I] := I * 2;
  // call
  X := Sum (List);
```

This is if you have a dynamic array. If you have a static array of the matching base type, you can also pass it to a functions expecting an open array parameter, or you can call the `slice` function to pass only a portion of the existing array (as indicated by its second parameter). The following snippet (also part of the `OpenArray` application project) shows how to pass a static array or a portion of it to the `Sum` function:

```
var
  List: array [1..10] of Integer;
  X, I: Integer;
begin
  // initialize the array
  for I := Low (List) to High (List) do
    List [I] := I * 2;

  // call
  X := Sum (List);
  Show (X.ToString);

  // pass portion of the array
  X := Sum (Slice (List, 5));
  Show (X.ToString);
```

Type-Variant Open Array Parameters

Besides these typed open array parameters, the Object Pascal language allows you to define type-variant or untyped open arrays. This special kind of array has an undefined number of elements, but also an undefined data type for those elements along with the possibility of passing elements of different types. This is one of the limited areas of the language that is not fully type safe.

Technically, the you can define a parameter of type `array of const` to pass an array with an undefined number of elements of different types to a function. For example, here is the definition of the `Format` function (we'll see how to use this function in Chapter 6, while covering strings, but I've already used it in some demos):

```
function Format (const Format: string;
  const Args: array of const): string;
```

The second parameter is an open array, which receives an undefined number of values. In fact, you can call this function in the following ways:

```
N := 20;
S := 'Total: ';
Show (Format ('Total: %d', [N]));
Show (Format ('Int: %d, Float: %f', [N, 12.4]));
Show (Format ('%s %d', [S, N * 2]));
```

Notice that you can pass a parameter as either a constant value, the value of a variable, or an expression. Declaring a function of this kind is simple, but how do you

142 - 05: Arrays and Records

code it? How do you know the types of the parameters? The values of a type-variant open array parameter are compatible with the `TVarRec` type elements.

note Do not confuse the `TVarRec` record with the `TVarData` record used by the `Variant` type. These two structures have a different aim and are not compatible. Even the list of possible types is different, because `TVarRec` can hold Object Pascal data types, while `TVarData` can hold Windows OLE data types. Variants are covered later in this chapter.

The following are the data types supported in a type-variant open array value and by the `TVarRec` record:

<code>vtInteger</code>	<code>vtBoolean</code>	<code>vtChar</code>
<code>vtExtended</code>	<code>vtString</code>	<code>vtPointer</code>
<code>vtPChar</code>	<code>vtObject</code>	<code>vtClass</code>
<code>vtWideChar</code>	<code>vtPWideChar</code>	<code>vtAnsiString</code>
<code>vtCurrency</code>	<code>vtVariant</code>	<code>vtInterface</code>
<code>vtWideString</code>	<code>vtInt64</code>	<code>vtUnicodeString</code>

The record structure has a field with the type (`VType`) and variant field you can use to access the actual data (more about records in a few pages, even if this is an advanced usage for that construct).

A typical approach is to use a case statement to operate on the different types of parameters you can receive in such a call. In the `SumAll` function example, I want to be able to sum values of different types, transforming strings to integers, characters to the corresponding ordinal value, and adding 1 for True Boolean values. The code is certainly quite advanced (and it uses pointer dereferences), so don't worry if you don't fully understand it for now:

```
function SumAll (const Args: array of const): Extended;
var
  I: Integer;
begin
  Result := 0;
  for I := Low(Args) to High (Args) do
    case Args [I].VType of
      vtInteger:
        Result := Result + Args [I].VInteger;
      vtBoolean:
        if Args [I].VBoolean then
          Result := Result + 1;
      vtExtended:
        Result := Result + Args [I].VExtended^;
      vtWideChar:
        Result := Result + Ord (Args [I].VWideChar);
      vtCurrency:
        Result := Result + Args [I].VCurrency^;
    end; // case
  end;
```

I've added this function to the `OpenArray` application project, which calls it as follows:

```
var
  X: Extended;
  Y: Integer;
begin
  Y := 10;
  X := SumAll ([Y * Y, 'k', True, 10.34]);
  Show ('SumAll: ' + X.ToString);
end;
```

The output of this call adds the square of `Y`, the ordinal value of `K` (which is 107), 1 for the Boolean value, and the extended number, resulting in:

```
| SumAll: 218.34
```

Record Data Types

While arrays define lists of identical items referenced by a numerical index, records define groups of elements of different types referenced by name. In other words, a record is a list of named items, or fields, each with a specific data type. The definition of a record type lists all these fields, giving each field a name used to refer to it. While in the early days of Pascal, records had only fields, now they can also have methods and operators, as we'll see in this chapter.

note Records are available in most programming languages. They are defined with the `struct` keyword in the C language, while C++ has an extended definition including methods, much like Object Pascal has. Some more “pure” object-oriented languages have only the notion of class, not that of a record or structure, but C# has recently reintroduced the concept.

Here is a small code snippet (from the `RecordsDemo` application project) with the definition of a record type, the declaration of a variable of that type, and few statements using this variable:

```
type
  TMyDate = record
    Year: Integer;
    Month: Byte;
    Day: Byte;
  end;

var
  BirthDay: TMyDate;
begin
```


144 - 05: Arrays and Records

```
BirthDay.Year := 1997;
BirthDay.Month := 2;
BirthDay.Day := 14;
Show ('Born in year ' + BirthDay.Year.ToString);
```

note The term records is at times used in a rather loose way to refer to two different elements of the language: a record type definition and a variable of record type (or record instance). Record is used as a synonym of both record type and record instance, unlike for class types in which case the instance is called object.

There is way more to this data structure in Object Pascal than a simple list of fields, as the remaining part of this chapter will illustrate, but let's start with this *traditional* approach to records. The memory for a record is generally allocated on the stack for a local variable and in the global memory for a global one. This is highlighted by a call to `SizeOf`, which returns the number of bytes required by a variable or type, like in this statement:

```
Show ('Record size is ' + SizeOf (BirthDay).ToString);
```

which returns 8 (why it does return 8 and not 6, 4 bytes for the Integer and two for each byte field, I'll discuss in the section “Fields Alignments”).

In other words, records are value types. This implies that if you assign a record to another, you are making a full copy. If you make a change to a copy, the original record won't be affected. This code snippets explains the concept in code terms:

```
var
  BirthDay: TMyDate;
  ADay: TMyDate;
begin
  BirthDay.Year := 1997;
  BirthDay.Month := 2;
  BirthDay.Day := 14;

  ADay := BirthDay;
  ADay.Year := 2008;

  Show (MyDateToString (BirthDay));
  Show (MyDateToString (ADay));
```

The output (in Japanese or international date format) is:

```
1997.2.14
2008.2.14
```

The same copy operation takes place when you pass a record as parameter to a function, like in the `MyDateToString` I used above:

```
function MyDateToString (MyDate: TMyDate): string;
begin
  Result := MyDate.Year.ToString + '.' +
    MyDate.Month.ToString + '.' +
```

```

    MyDate.Day.ToString;
end;

```

Each call to this function involves a complete copy of the record's data. To avoid the copy, and to possibly make a change to the original record you have to explicitly use a reference parameter. This is highlighted by the following procedure, that makes some changes to a record passed as parameter:

```

procedure IncreaseYear (var MyDate: TMyDate);
begin
    Inc (MyDate.Year);
end;

var
    ADay: TMyDate;
begin
    ADay.Year := 2020;
    ADay.Month := 3;
    ADay.Day := 18;

    Increaseyear (ADay);
    Show (MyDateToString (ADay));

```

Given the `Year` field of the original record value is increased by the procedure call, the final output is one year later than the input:

```

2021.3.18

```

Using Arrays of Records

As I mentioned, arrays represent a data structure repeated several times, while records a single structure with different elements. Given these two type constructors are orthogonal, it is very common to use them together, defining arrays of records (while it is possible but uncommon to see records of arrays).

The array code is just like that of any other array, with each array element taking the size of the specific record type. While we'll see later how to use more sophisticated collection or container classes (for lists of elements), there is a lot in terms of data management you can achieve with arrays of records.

In the `RecordsTest` application project I've added an array of the `TMyDate` type, which can be allocated, initialized and used with code like the following:

```

var
    DatesList: array of TMyDate;
    I: Integer;
begin
    // allocate array elements
    SetLength (DatesList, 5);

```

146 - 05: Arrays and Records

```
// assign random values
for I := Low(DatesList) to High(DatesList) do
begin
    DatesList[I].Year := 2000 + Random (50);
    DatesList[I].Month := 1 + Random (12);
    DatesList[I].Day := 1 + Random (27);
end;

// display the values
for I := Low(DatesList) to High(DatesList) do
    Show (I.ToString + ': ' +
        MyDateToString (DatesList[I]));
```

Given the app uses random data, the output will be different every time, and could be like the following I've captured:

```
0: 2014.11.8
1: 2005.9.14
2: 2037.9.21
3: 2029.3.12
4: 2012.7.2
```

note Records in an array can be automatically initialized in case of a managed record, a feature introduced in Delphi 10.4 Sydney that I'll cover later in this chapter.

Variant Records

Since the early versions of the language, record types can also have a variant part; that is, multiple fields can be mapped to the same memory area, even if they have a different data type. (This corresponds to a *union* in the C language.) Alternatively, you can use these variant fields or groups of fields to access the same memory location within a record, but considering those values from different perspectives (in terms of data types). The main uses of this type were to store similar, but different data and to obtain an effect similar to that of typecasting (something used in the early days of the language, when direct typecasting was not allowed). The use of variant record types has been largely replaced by object-oriented and other modern techniques, although some system libraries use them internally in special cases.

The use of a variant record type is not type-safe and is not a recommended programming practice, particularly for beginners. You won't need to tackle them until you are really an Object Pascal expert, anyway... and that's why I decided to avoid showing you actual samples and covering this feature in more detail. If you really want a hint, have a look at the use of `TVarRec` I did in the demo of the section "Type-Variant Open Array Parameters".

Fields Alignments

Another advanced topic related with records is the way their fields are aligned, which also helps understand the actual size of a record. If you look into libraries, you'll often see the use of the `packed` keyword applied to records: this implies the record should use the minimum possible amount of bytes, even if this result in slower data access operations.

The difference is traditionally related to 16-bit or 32-bit alignment of the various fields, so that a byte followed by an integer might end up taking up 32 bits even if only 8 are used. This is because accessing the following integer value on the 32-bit boundary makes the code faster to execute.

note The fields size and alignment depends on the size of the type. For any type with a size not a power of 2 (or 2^N) the size is the next higher power of 2. So for example the `Extended` type, which uses 10 bytes, in a record takes 16 bytes (unless the record is `packed`).

In general field alignment is used by data structures like records to improve the access speed to individual fields for some CPU architectures. There are different parameters you can apply to the `$ALIGN` compiler directive to change it.

With `{ $ALIGN 1 }` the compiler will save on memory usage by using all possible bytes, like when you use the `packed` specifier for a record. At the other extreme, the `{ $ALIGN 16 }` will use the largest alignment. Further options use 4 and 8 alignments.

As an example, if I go back to the `RecordsTest` project and add the keyword `packed` to the record definition:

```
type
  TMyDate = packed record
    Year: Integer;
    Month: Byte;
    Day: Byte;
  end;
```

the output to the call `SizeOf` will now return 6 rather than 8.

As a more advanced example, which you can skip if you are not already a fluent Object Pascal developer, let's consider the following structure (available in the `AlignTest` application project):

```
type
  TMyRecord = record
    c: Byte;
    w: Word;
    b: Boolean;
    i: Integer;
    d: Double;
```

148 - 05: Arrays and Records

```
|   end;
```

With `{ $ALIGN 1 }` the structure takes 16 bytes (the value returned by `sizeof`) and the fields will be at the following relative memory addresses:

```
| c: 0 w: 1 b: 3 i: 4 d: 8
```

note Relative addresses are computed by allocating the record and computing the difference between the numeric value of a pointer to the structure and that of a pointer to the given field, with an expression like: `UIntPtr(@MyRec.w) - UIntPtr(@MyRec.l)`. The concept of pointers and the address of (`@`) operator are covered later in this chapter.

In contrast, if you change the alignment to 4 (which can lead to optimized data access) the size will be 20 bytes and the relative addresses:

```
| c: 0 w: 2 b: 4 i: 8 d: 12
```

If you go to the extreme option and use `{ $ALIGN 16 }`, the structure requires 24 bytes and maps the fields as follow:

```
| c: 0 w: 2 b: 4 i: 8 d: 16
```

What About the With Statement?

Another traditional language statement used to work with records or classes, is the `with` statement. This keyword used to be peculiar to the Pascal syntax, but it was later introduced in JavaScript and Visual Basic. This is a keyword that can come up very handy to write less code, but it can also become very dangerous as it makes code far less readable.

You'll find a lot of debate around the `with` statement, and I tend to agree this should be used sparingly, if at all. In any case, I felt it was important to include it in this book anyway (differently from `goto` statements).

note There is some debate about whether it will make sense to remove `goto` statements from the Object Pascal language, and it was also discussed whether to remove `with` from the mobile version of the language. While there are some legitimate usages, given the scoping problems `with` statements can cause, there are good reasons to discontinue this features (or change it so that an alias name is required as in C#).

The `with` statement is nothing but a shorthand. When you need to refer to a record type variable (or an object), instead of repeating its name every time, you can use a `with` statement.

For example, while presenting the record type, I wrote this code:

```

var
  BirthDay: TMyDate;
begin
  BirthDay.Year := 2008;
  BirthDay.Month := 2;
  BirthDay.Day := 14;

```

Using a `with` statement, I could modify the final part of this code, as follows:

```

with BirthDay do
begin
  Year := 2008;
  Month := 2;
  Day := 14;
end;

```

This approach can be used in Object Pascal programs to refer to components and other classes. When you work with components or classes in general, the `with` statement allows you to skip writing some code, particularly for nested data structures.

So, why am I not encouraging the use of the `with` statement? The reason is it can lead to subtle errors that are very hard to capture. While some of these hard-to-find errors are not easy to explain at this point of the book, let's consider a mild scenario, that can still lead to you scratching your head. This is a record type and some code using it:

```

type
  TMyRecord = record
    MyName: string;
    MyValue: Integer;
  end;

procedure TForm1.Button2Click(Sender: TObject);
var
  Record1: TMyRecord;
begin
  with Record1 do
  begin
    with Record1 do
    begin
      MyName := 'Joe';
      MyValue := 22;
    end;

    with Record1 do
      Show (Name + ': ' + MyValue.ToString);
    end;
  end;
end;

```

Right? The application compiles and runs, but its output is not what you might expect (at least at first sight):

```
Form1: 22
```

The string part of the output is not the record value that was set earlier. The reason is that the second `with` statement erroneously uses the `Name` field, which is not the

150 - 05: Arrays and Records

record field but another field that happens to be in scope (specifically the name of the form object the `Button2Click` method is part of).

If you had written:

```
| Show (Record1.Name + ':' + Record1.MyValue.ToString);
```

the compiler would have shown an error message, indicating the given record structure hasn't got a `Name` field.

In general, we can say that since the `with` statement introduces new identifiers in the current scope, we might hide existing identifiers, or wrongfully access another identifier in the same scope. This is a good reason for discouraging the use of the `with` statement. Even more you should avoid using multiple `with` statements, such as:

```
| with MyRecord1, MyDate1 do...
```

The code following this would probably be highly unreadable, because for each field used in the block you would need to think about which record it refers to.

Records with Methods

In Object Pascal records are more powerful than in the original Pascal language or than structs are in the C language. Records, in fact, can have procedure and functions (called methods) associated with them. They can even redefine the language operators in custom ways (a feature called *operator overloading*), as you'll see in the next section.

A record with methods is somewhat similar to a class, as we'll find out later, with the most important difference being the way these two structures manage memory.

Records in Object Pascal have two fundamental features of modern programming languages:

- **Methods**, which are functions and procedures connected with the record data structure and having direct access for the record fields. In other words, methods are function and procedures declared (or having a forward declaration) within the record type definition.
- **Encapsulation**, which is the ability to hide direct access for some of the fields (or methods) of a data structure from the rest of the code. You can obtain encapsulation using the `private` access specifier, while fields and methods visible to the outside as marked as `public`. The default specifier for a record is `public`.

Now that you have the core concepts around extended records, let's look at the definition of a sample record, taken from the `RecordMethods` application project:

```
type
  TMyRecord = record
    private
      Name: string;
      Value: Integer;
      SomeChar: Char;
    public
      procedure Print;
      procedure SetValue (NewString: string);
      procedure Init (NewValue: Integer);
    end;
```

You can see the record structure is divided in two parts, private and public. You can have multiple sections, as the private and public keywords can be repeated as many times as you want, but a clear division of these two sections certainly helps readability. The methods are listed in the record definition (like in a class definition) without their complete code. In other words, it has a forward declaration of the method.

How do you write the actual code of a method, its complete definition? Almost in the same way you'd code a global function or procedure. The differences are in the way you write the method name, which is a combination of the record type name and the actual record name and on the fact you can directly refer to the fields and the other methods of the record directly, with no need to write the name of the record:

```
procedure TMyRecord.SetValue (NewString: string);
begin
  Name := NewString;
end;
```

tip While it might seem tedious having to write the definition of the method first and its full declaration next, you can use the `Ctrl+Shift+C` combination in the IDE editor to generate one from the other automatically. Also you can use the `Ctrl+Shift+Up/Down Arrow` keys to move from a method declaration to the corresponding definition and vice versa.

Here is the code of the other methods of this record type:

```
procedure TMyRecord.Init(NewValue: Integer);
begin
  Value := NewValue;
  SomeChar := 'A';
end;

function TMyRecord.ToString: string;
begin
  Result := Name + ' [' + SomeChar + ']: ' + Value.ToString;
end;
```


152 - 05: Arrays and Records

Here is a sample snippet of how you can use this record:

```
var
  MyRec: TMyRecord;
begin
  MyRec.Init(10);
  MyRec.SetValue ('hello');
  Show (MyRec.ToString);
```

As you might have guessed, the output will be:

```
| hello [A]: 10
```

Now what if you want to use the fields from the code that uses the record, like in the snippet above:

```
| MyRec.Value := 20;
```

This actually compiles and works, which might be surprising as we declared the field in the private section, so that only the record methods can access it.

The truth is that in Object Pascal the `private` access specifier is actually enabled only between different units, so that line wouldn't be legal in a different unit, but can be used in the unit that originally defined the data type. As we will see, this is also true for classes.

Self: The Magic Behind Records

Suppose you have two records, like `myrec1` and `myrec2` of the same record type. When you call a method and execute its code, how does the method know which of the two copies of the record it has to work with? Behind the scenes, when you define a method the compiler adds a hidden parameter to it, a reference to the record you have applied the method to.

In other words, the call to the method above is converted by the compiler in something like:

```
// you write
MyRec.SetValue ('hello');

// the compiler generates
SetValue (@MyRec, 'hello');
```

In this pseudo code, the `@` is the *address of* operator, used to get the memory location of a record instance.

note Again, the *address of* operator is shortly covered at the end of this chapter in the (advanced) section titled “What About Pointers?”

This is how the calling code is translated, but how can the actual method call refer and use this hidden parameter? By implicitly using a special keyword called `self`. So the method's code could be written as:

```
procedure TMyRecord.SetValue (NewString: string);
begin
    self.Name := NewString;
end;
```

While this code compiles, it makes little sense to use `self` explicitly, unless you need to refer to the record as a whole, for example passing the record as parameter to another function. This happens more frequently with classes, which have the same exact hidden parameter for methods and the same `self` keyword.

One situation in which using an explicit `self` parameter can make the code more readable (even if it is not required) is when you are manipulating a second data structure of the same type, as in case you are testing a value from another instance:

```
function TMyRecord.IsSameName (ARecord: TMyRecord): Boolean;
begin
    Result := (self.Name = ARecord.Name);
end;
```

note The “hidden” `self` parameter is called `this` in C++ and Java, but it is called `self` in Objective-C (and in Object Pascal, of course).

Initializing Records

When you define a variable of a record type (or a record instance) as a global variable its fields are initialized, but when you define one on the stack (as a local variable of a function or procedure, it isn't). So if you write code like this (also part of the `RecordMethods` project):

```
var
    MyRec: TMyRecord;
begin
    Show (MyRec.ToString);
```

its output will be more or less random. While the string is initialized to an empty string, the character field and the integer field will have the data that happened to be at the given memory location (just as it happens in general for a character or integer variable on the stack). In general, you'd get different output depending on the actual compilation or execution, such as:

```
| []: 1637580
```

154 - 05: Arrays and Records

That's why it is important to initialize a record (as most other variables) before using it, to avoid the risk of reading illogical data, which can even potentially crash the application.

There are two radically different approaches to handle this scenario. The first is the use of constructors for record, as covered in the next section. The second is the use of managed records, a new feature in Delphi 10.4 that I'll cover later in this chapter.

Records and Constructors

Let's start with regular constructors. Records support a special type of methods called constructors, that you can use to initialize the record data. Differently from other methods, constructors can also be applied to a record type to define a new instance (but they can still be applied to an existing instance).

This is how you can add a constructor to a record:

```
type
  TMyNewRecord = record
  private
    ...
  public
    constructor Create (NewString: string);
    function ToString: string;
    ...
```

The constructor is a method with code:

```
constructor TMyNewRecord.Create (NewString: string);
begin
  Name := NewString;
  Init (0);
end;
```

Now you can initialize a record with either of the two following coding styles:

```
var
  MyRec, MyRec2: TMyNewRecord;
begin
  MyRec := TMyNewRecord.Create ('Myself'); // class-like
  MyRec2.Create ('Myself'); // direct call
```

Notice that record constructors must have parameters: If you try with `Create()` you'll get the error message "Parameterless constructors not allowed on record types".

note According to the documentation the definition of a parameterless constructor for records is reserved for the system (which has its way to initialize some of the records fields, such as strings and interfaces). This is why any user defined constructor must have at least one parameter. Of course, you can also have multiple overloaded constructors or multiple constructors with different names. I'll cover this in more detail when discussing constructors for classes. Managed records, as we'll see shortly, use a different syntax and do not introduce parameterless constructors, but rather an `Initialize` class method.

Operators Gain New Ground

Another Object Pascal language feature related with records is operator overloading; that is, the ability to define your own implementation for standard operations (addition, multiplication, comparison, and so on) on your data types. The idea is that you can implement an add operator (a special `Add` method) and then use the `+` sign to call it. To define an operator you use `class operator` keyword combination.

note By reusing existing reserved words, the language designers managed to have no impact on existing code. This is something they've done quite often recently in keyword combinations like `strict private`, `class operator`, and `class var`.

The term *class* here relates to class methods, a concept we'll explore much later (in Chapter 12). After the directive you write the operator's name, such as `Add`:

```
type
  TPointRecord = record
  public
    class operator Add (
      A, B: TPointRecord): TPointRecord;
```

The operator `Add` is then called with the `+` symbol, as you'd expect:

```
var
  A, B, B: TPointRecord;
begin
  ...
  B := A + B;
```

So which are the available operators? Basically the entire operator set of the language, as you cannot define brand new language operators:

- **Cast Operators:** `Implicit` and `Explicit`
- **Unary Operators:** `Positive`, `Negative`, `Inc`, `Dec`, `LogicalNot`, `BitwiseNot`, `Trunc`, and `Round`
- **Comparison Operators:** `Equal`, `NotEqual`, `GreaterThan`, `GraterThanOrEqual`, `LessThan`, and `LessThenOrEqual`

156 - 05: Arrays and Records

- **Binary Operators:** Add, Subtract, Multiply, Divide, IntDivide, Modulus, ShiftLeft, ShiftRight, LogicalAnd, LogicalOr, LogicalXor, BitwiseAnd, BitwiseOr, and BitwiseXor.
- **Managed Record Operators:** Initialize, Finalize, Assign (see the following section “Operators and Custom Managed Record” for specific information on these 3 operators added in Delphi 10.4)

In the code calling the operator, you do not use these names but use the corresponding symbol. You use these special names only in the definition, with the `class` operator prefix to avoid any naming conflict. For example, you can have a record with an `Add` method and add an `Add` operator to it.

When you define these operators, you spell out the parameters, and the operator is applied only if the parameters match exactly. To add two values of different types, you'll have to specify two different `Add` operations, as each operand could be the first or second entry of the expression. In fact, the definition of operators provides no automatic commutativity. Moreover, you have to indicate the type very precisely, as automatic type conversions don't apply. Many times this implies defining multiple overloaded versions the operator, with different types of parameters.

Another important factor to notice is that there are two special operators you can define for data conversion, `Implicit` and `Explicit`. The first is used to define an implicit type cast (or silent conversion), which should be perfect and not lossy. The second, `Explicit`, can be invoked only with an explicit type cast from a variable of a type to another given type. Together these two operators define the casts that are allowed to and from the given data type.

Notice that both the `Implicit` and the `Explicit` operators can be overloaded based on the function return type, which is generally not possible for overloaded methods. In case of a type cast, in fact, the compiler knows the expected resulting type and can figure out which is the typecast operation to apply. As an example, I've written the `operatorsOver` application project, which defines a record with a few operators:

```
type
  TPointRecord = record
  private
    X, Y: Integer;
  public
    procedure SetValue (X1, Y1: Integer);
    class operator Add (A, B: TPointRecord): TPointRecord;
    class operator Explicit (A: TPointRecord): string;
    class operator Implicit (X1: Integer): TPointRecord;
  end;
```

Here is the implementation of the methods of the record:

```

class operator TPointRecord.Add(
  A, B: TPointRecord): TPointRecord;
begin
  Result.X := A.X + B.X;
  Result.Y := A.Y + B.Y;
end;

class operator TPointRecord.Explicit(
  A: TPointRecord): string;
begin
  Result := Format('%d:%d', [A.X, A.Y]);
end;

class operator TPointRecord.Implicit(
  X1: Integer): TPointRecord;
begin
  Result.X := X1;
  Result.Y := 10;
end;

```

Using such a record is quite straightforward, as you can write code like this:

```

procedure TForm1.Button1Click(Sender: TObject);
var
  A, B, C: TPointRecord;
begin
  A.SetValue(10, 10);
  B := 30;
  C := A + B;
  Show (string(C));
end;

```

The second assignment (`B:=30;`) is done using the implicit operators, due to the lack of a cast while the `Show` call uses the cast notation to activate an explicit type conversion. Consider also that the operator `Add` doesn't modify its parameters; rather it returns a brand new value.

note The fact operators return new values is what make it harder to think of operator overloading for classes. If the operator creates a new temporary objects who is going to dispose it?

Operators Overloading Behind the Scenes

This is a fairly advanced, short section, you might want to skip at first reading

A little known fact is that it is technically possible to call an operator using its fully qualified internal name (like `&&op_Addition`), prefixing it with a double `&`, instead of using the operator symbol. For example, you can rewrite the records sum as follows (see the demo for the complete listing):

```

| C := TPointRecord.&&op_Addition(A, B);

```

158 - 05: Arrays and Records

although I can see very few marginal cases in which you might want to do so. (The entire purpose of defining operators is to be able to use a friendlier notation than a method name, not an uglier one as the preceding direct call.)

Implementing Commutativity

Suppose you want implement the ability to add an integer number to one of your records. You can define the following operator (that is available in the code of the `operatorsOver` application project, for a slightly different record type):

```
class operator TPointRecord2.Add (A: TPointRecord2;  
    B: Integer): TPointRecord2;  
begin  
    Result.X := A.X + B;  
    Result.Y := A.Y + B;  
end;
```

note The reason I've defined this operator on a new type rather than the existing one is that the same structure already defines an `Implicit` conversion of an integer to the record type, so I can already add integers and records without defining a specific operator. This issue is explained better in the next section.

Now you can legitimately add a floating point value to a record:

```
var  
    A: TPointRecord2;  
begin  
    A.SetValue(10, 20);  
    A := A + 10;
```

However if you try to write the opposite addition:

```
| A := 30 + A;
```

this will fail with the error:

```
| [dcc32 Error] E2015 Operator not applicable to this operand type
```

In fact, as I mentioned, commutativity is not automatic for operators applied to variables of different types, but must be specifically implemented either repeating the call or calling (like below) the other version of the operator:

```
class operator TPointRecord2.Add(B: Integer;  
    A: TPointRecord2): TPointRecord2;  
begin  
    Result := A + B; // implement commutativity  
end;
```

Implicit Cast and Type Promotions

It is important to notice that the rules related to the resolution of calls involving operators are different from the traditional rules involving methods. With automatic type promotions there's the chance that a single expression will end up calling different versions of an overloaded operator and cause ambiguous calls. This is why you need to take a lot of care when writing `Implicit` operators.

Consider these expressions from the previous example:

```
A := 50;
C := A + 30;
C := 50 + 30;
C := 50 + TPointRecord(30);
```

They are all legal! In the first case, the compiler converts 30 to the proper record type, in the second the conversion takes place after the assignment, and in the third the explicit cast forces an implicit one on the first value, so that the addition being performed is the custom one among records. In other words the result of the second operation is different from the other two, as highlighted in the output and in the expanded version of these statements:

```
// output
(80:20)
(80:10)
(80:20)

// expanded statements
C := A + TPointRecord(30);
// that is: (50:10) + (30:10)

C := TPointRecord (50 + 30);
// that is 80 converted into (80:10)

C := TPointRecord(50) + TpointRecord(30);
// that is: (50:10) + (30:10)
```

Operators and Custom Managed Record

There is a special set of operators you can use for records in the Delphi language to define a custom managed record. Before we get there, let me recap the rules for records memory initialization, and the difference between plain records and managed records.

Records in Delphi can have fields of any data type. When a record has plain (non-managed) fields, like numeric or other enumerated values there isn't much to do for the compiler. Creating and disposing the record consists of allocating memory or

160 - 05: Arrays and Records

getting rid of the memory location. (Notice that by default Delphi does not zero-initialize records.)

If a record has a field of a type managed by the compiler (like a string or an interface), the compiler needs to inject extra code to manage the initialization or finalization. A string, for example, is reference counted so when the record goes out of scope the string inside the record needs to have its reference count decreased, which might lead to deallocating the memory for the string. Therefore, when you are using such a managed record in a section of the code, the compiler automatically adds a try-finally block around that code, and makes sure the data is cleared even in case of an exception. This has been the case for a long time. In other words, managed records have been part of the Delphi language.

Now starting with 10.4 the Delphi record type supports custom initialization and finalization, beyond the default operations the compiler does for managed records. You can declare a record with custom initialization and finalization code regardless of the data type of its fields, and you can write such custom initialization and finalization code. These records are indicated as “*custom managed records*”.

A developer can turn a record into a custom managed record by adding one of the specific new operators to the record type, or more than one:

- Operator `Initialize` is invoked after the memory for the record has been allocated, allowing you to write code to set an initial value for the fields
- Operator `Finalize` is invoked before the memory for the record is de-allocated and allows you to do any required cleanup
- Operator `Assign` is invoked when a record data is copied into another record of the same type, so you can copy the information from one record to the other in specific, custom ways

note Given that managed records finalization is executed even in case of an exception (with the compiler automatically generating a try-finally block), they are often used as an alternative way to protect resource allocation or implement cleanup operations. We'll see an example of this usage in section “Restore the Cursor with a Managed Record” of Chapter 9.

Records with Initialize and Finalize Operators

Let's start with initialization and finalization. Below is a simple code snippet:

```
type
  TMyRecord = record
    Value: Integer;
    class operator Initialize (out Dest: TMyRecord);
    class operator Finalize (var Dest: TMyRecord);
  end;
```

You need to write the code for the two class methods, of course, for example logging their execution or initializing the record value. In this example (part of the `ManagedRecords_101` demo project) I'm also logging a reference to memory location, to see which record is performing each individual operation:

```
class operator TMyRecord.Initialize (out Dest: TMyRecord);
begin
    Dest.Value := 10;
    Log('created' + IntToHex (Integer(Pointer(@Dest))));
end;

class operator TMyRecord.Finalize(var Dest: TMyRecord);
begin
    Log('destroyed' + IntToHex (Integer(Pointer(@Dest))));
end;
```

The difference between this construction mechanism and what was previously available for records is the automatic invocation. If you write something like the code below, you can invoke both the initialization and the finalization code and end up with a try-finally block generated by the compiler for your managed record instance:

```
procedure LocalVarTest;
var
    My1: TMyRecord;
begin
    Log (My1.Value.ToString);
end;
```

With this code you'll get a log like:

```
created 0019F2A8
10
destroyed 0019F2A8
```

Another scenario is the use of inline variables, like in:

```
begin
    var T: TMyRecord;
    Log(T.Value.ToString);
```

which gets you the same sequence in the log.

The Assign Operator

In general, the `:=` assignment flatly copies all of the data of the record fields. For a managed records (with strings and other managed types) those are treated properly by the compiler.

When you have custom data fields and custom initialization you might want to change the default behavior. This is why for custom managed records you can also define an assignment operator. The new operator is invoked with the `:=` syntax, but defined as `Assign`:

162 - 05: Arrays and Records

```
class operator Assign (var Dest: TMyRecord;  
    const [ref] Src: TMyRecord);
```

The operator definition must follow very precise rules, including having the first parameter as a parameter passed by reference (**var**), and the second as a **const** parameter passed by reference. If you fail to do so, the compiler issues error messages like the following:

```
[dcc32 Error] E2617 First parameter of Assign operator must be a var  
parameter of the container type  
[dcc32 Hint] H2618 Second parameter of Assign operator must be a  
const[Ref] or var parameter of the container type
```

There is a sample case invoking the Assign operator

```
var  
    My1, My2: TMyRecord;  
begin  
    My1.Value := 22;  
    My2 := My1;
```

which produces this log (in which I also add a sequence number to the record):

```
created 5 0019F2A0  
created 6 0019F298  
5 copied to 6  
destroyed 6 0019F298  
destroyed 50019F2A0
```

Notice that the sequence of destruction is reversed from the sequence of construction, with the last record created being the first destroyed.

Passing Managed Records as Parameters

Managed records can work differently from regular records also when passed as parameters or returned by a function. Here are several routines showing the various scenarios:

```
procedure ParByValue (Rec: TMyRecord);  
procedure ParByConstValue (const Rec: TMyRecord);  
procedure ParByRef (var Rec: TMyRecord);  
procedure ParByConstRef (const [ref] Rec: TMyRecord);  
function ParReturned: TMyRecord;
```

Now without going over each log one by one (you can see them by running the ManagedRecords_101 demo), this is the summary of the information:

- ParByValue creates a new record and calls the assignment operator (if available) to copy the data, destroying the temporary copy when exiting the procedure
- ParByConstValue makes no copy, and no call at all

- `ParByRef` makes no copy, no call
- `ParByConstRef` makes no copy, no call
- `ParReturned` creates a new record (via `Initialize`) and on return it calls the `Assign` operator, if the call is like `my1 := ParReturned`, and deletes the temporary record once assigned.

Exceptions and Managed Records

When an exception is raised, records in general are cleared even when no explicit `try`, `finally` block is present, unlike objects. This is a fundamental difference and key to the real usefulness of managed records.

```

procedure ExceptionTest;
begin
  var A: TMRE;
  var B: TMRE;
  raise Exception.Create('Error Message');
end;

```

Within this procedure, there are two constructor calls and two destructor calls. Again, this is a fundamental difference and a key feature of managed records.

Arrays of Managed Records

If you define a static array of managed records, they are initialized calling the `Initialize` operator at the point declaration:

```

var
  A1: array [1..5] of TMyRecord; // call here
begin
  Log ('ArrOfRec');

```

They are all destroyed when they get out of scope. If you define dynamic array of managed records, the initialization code is called with the array is sized (with `SetLength`):

```

var
  A2: array of TMyRecord;
begin
  Log ('ArrOfDyn');
  SetLength(A2, 5); // call here

```

Variants

Originally introduced in the language to provide full Windows OLE and COM support, Object Pascal has the concept of a *loosely typed* native data type called `variant`. Although the name reminds of variant records (mentioned earlier) and the implementation has some similarity with open array parameters, this is a separate feature with a very specific implementation (uncommon in languages outside of the Windows development world).

In this section I won't really refer to OLE and other scenarios in which this data type is used (like fields access for data sets), I only want to discuss this data type from a general perspective.

I'll get back to dynamic types, RTTI, and reflection in Chapter 16, where I'll also cover a related (but type safe and much faster) RTL data type called `TValue`.

Variants Have No Type

In general, you can use a variable of the variant type to store any of the basic data types and perform numerous operations and type conversions. Automatic type conversions go against the general type-safe approach of the Object Pascal language and is an implementation of a type of dynamic typing originally introduced by languages like Smalltalk and Objective-C, and recently made popular in scripting languages including JavaScript, PHP, Python, and Ruby.

A variant is type-checked and computed at run time. The compiler won't warn you of possible errors in the code, which can be caught only with extensive testing. On the whole, you can consider the code portions that use variants to be interpreted code, because, as with interpreted code, many operations cannot be resolved until run time. In particular this affects the speed of the code.

Now that I've warned you against the use of the `variant` type, it is time to look at what you can do with it. Basically, once you've declared a variant variable such as the following:

```
var
  v: variant;
```

you can assign values of several different types to it:

```
v := 10;
v := 'Hello, world';
v := 45.55;
```

Once you have the variant value, you can copy it to any compatible or incompatible data type. If you assign a value to an incompatible data type, the compiler will generally not flag it with an error, but will perform a runtime conversion if this makes sense. Otherwise it will issue a run-time error. Technically a variant stores type information along with the actual data, allowing a number of handy, but slow and unsafe, run-time operations.

Consider the following code (part of the `variantTest` application project), which is an extension of the code above:

```
var
  v: variant;
  s: string;
begin
  v := 10;
  s := v;
  v := v + s;
  show (v);

  v := 'Hello, world';
  v := v + s;
  show (v);

  v := 45.55;
  v := v + s;
  show (v);
```

Funny, isn't it? This is the output (not surprisingly):

```
20
Hello, world10
55.55
```

Besides assigning a variant holding a string to the `s` variable, you can assign to it a variant holding an integer or a floating-point number. Even *worse*, you can use the variants to compute values, with the operation `v := v + s`; that gets interpreted in different ways depending on the data stored in the variant. In the code above, that same line can add integers, floating point values, or concatenate strings.

Writing expressions that involve variants is risky, to say the least. If the string contains a number, everything works. If not, an exception is raised. Without a compelling reason to do so, you shouldn't use the `variant` type; stick with the standard Object Pascal data types and type-checking approach.

Variants in Depth

For those interested in understanding variants in more details, let me add some technical information about how variants work and how you can have more control

166 - 05: Arrays and Records

on them. The RTL includes a variant record type, `TVarData`, which has the same memory layout as the `variant` type. You can use this to access the actual type of a variant. The `TVarData` structure includes the type of the Variant, indicated as `vType`, some reserved fields, and the actual value.

note For more details look to the `TVarData` definition in the RTL source code, in the `System` unit. This is far from a simple structure and I recommend only developers with some experience look into the implementation details of the variant type.

The possible values of the `vType` field correspond to the data types you can use in OLE automation, which are often called OLE types or *variant types*. Here is a complete alphabetical list of the available variant types:

<code>varAny</code>	<code>varArray</code>	<code>varBoolean</code>
<code>varByte</code>	<code>varByRef</code>	<code>varCurrency</code>
<code>varDate</code>	<code>varDispatch</code>	<code>varDouble</code>
<code>varEmpty</code>	<code>varError</code>	<code>varInt64</code>
<code>varInteger</code>	<code>varLongWord</code>	<code>varNull</code>
<code>varOLEStr</code>	<code>varRecord</code>	<code>varShortInt</code>
<code>varSingle</code>	<code>varSmallInt</code>	<code>varString</code>
<code>varTypeMask</code>	<code>varUInt64</code>	<code>varUnknown</code>
<code>varUString</code>	<code>varVariant</code>	<code>varWord</code>

Most of these constant names of variant types are easy to understand. Notice that there is the concept of *null value*, you obtain by assigning `NULL` (and not `nil`).

There are also many functions for operating on variants that you can use to make specific type conversions or to ask for information about the type of a variant (see, for example, the `varType` function). Most of these type conversion and assignment functions are actually called automatically when you write expressions using variants. Other variant support routines actually operate on variant arrays, again a structure used almost exclusively for OLE integration on Windows.

Variants Are Slow

Code that uses the `variant` type is slow, not only when you convert data types, but even when you simply add two variant values holding integers. They are almost as slow as interpreted code. To compare the speed of an algorithm based on variants with that of the same code based on integers, you can look at the second button of the `variantTest` project.

This program runs a loop, timing its speed and showing the status in a progress bar. Here is the first of the two very similar loops, based on `Int64` and variants:

```

const
  maxno = 10000000;  // 10 million

var
  time1, time2: TDateTime;
  n1, n2: Variant;
begin
  time1 := Now;
  n1 := 0;
  n2 := 0;

  while n1 < maxno do
    begin
      n2 := n2 + n1;
      Inc (n1);
    end;

  // we must use the result
  time2 := Now;
  Show (n2);
  Show ('Variants: ' + FormatDateTime (
    'ss.zzz', Time2-Time1) + ' seconds');

```

The timing code is worth looking at, because it's something you can easily adapt to any kind of performance test. As you can see, the program uses the `Now` function to get the current time and the `FormatDateTime` function to output the time difference, showing only the seconds ("ss") and the milliseconds ("zzz").

In this example the speed difference is actually so great that you'll notice it even without precise timing:

```

49999995000000
Variants: 01.169 seconds
49999995000000
Integers: 00.026 second

```

These are the numbers I get on my Windows virtual machine, and that's about 50 times slower for the variant based code. The actual values depend on the computer you use to run this program, but the relative difference won't change much. Even on my Android phone I get a similar proportion (but much longer times overall):

```

49999995000000
Variants: 07.717 seconds
49999995000000
Integers: 00.157 second

```

On my phone this code takes 6 times as much as on Windows, but now the fact is the net different is over 7 seconds, making the variant based implementation noticeably slow to the user, while the `Int64` based one is still extremely fast (a user would hardly notice a tenth of a second).

What About Pointers?

Another fundamental data type of the Object Pascal language is represented by pointers. Some of the object-oriented languages have gone a long way to hide this powerful, but dangerous, language construct, while Object Pascal lets a programmer use it when needed (which is generally not very often).

But what is a pointer, and where does its name come from? Differently than most other data types, a pointer doesn't hold an actual value, but it holds an indirect reference to a variable, which in turn has a value. A more technical way to express this is that a pointer type defines a variable that holds the memory address of another variable of a given data type (or of an undefined type).

note This is an advanced section of the book, added here because pointers are part of the Object Pascal language and should be part of the core knowledge of any developer, although it is not a basic topic and if you are new to the language you might want to skip this section the first time you read the book. Again, there is a chance you might have used programming languages with no (explicit) pointers, so this short section could be an interesting read!

The definition of a pointer type is not based on a specific keyword, but uses a special symbol, the caret (^). For example you can define a type representing a pointer to variable of the Integer type with the following declaration:

```
type
  TPointerToInt = ^Integer;
```

Once you have defined a pointer variable, you can assign to it the address of another variable of the same type, using the @operator:

```
var
  P: ^Integer;
  X: Integer;
begin
  X := 10;
  P := @X;
  // change the value of X using the pointer
  P^ := 20;
  Show ('X: ' + X.ToString);
  Show ('P^: ' + P^.ToString);
  Show ('P: ' + UIntPtr(P).ToHexString (8));
```

This code is part of the `PointersTest` application project. Given the pointer `P` refers to the variable `X`, you can use `P^` to refer to the value of the variable, and read or change it. You can also display the value of the pointer itself, that is the memory address of `X`, by casting the pointer to a number using the special type `UIntPtr` (see the note below for more information). Rather than showing the plain numeric value,

the code shows the hexadecimal representation, which is more common for memory addresses. This is the output (where the pointer address might depend on the specific compilation):

```
X: 20
P^: 20
P: 0018FC18
```

warn Casting a pointer to an Integer is correct code only on 32-bit platforms when limited to 2GB. If you enable the use of a larger memory space, you'd have to use the Cardinal type. For 64-bit platforms a better option would be to use `NativeUInt`. However, there is an alias of this type, specifically meant for pointers and called `UIntPtr`, which is the best option for this scenario as using it you clearly indicate your intentions to the Delphi compiler.

Let me summarize, for clarity. When you have a pointer `P`:

- By using the pointer directly (with the expression `P`) you refer to the address of the memory location the pointer is referring to
- By dereferencing the pointer (with the expression `P^`) you refer to the actual content of that memory location

Instead of referring to an existing memory location, a pointer can also refer to a new and specific memory block dynamically allocated on the heap with the `New` procedure. In this case, when you don't need the value accessed by the pointer anymore, you'll also have to get rid of the memory you've dynamically allocated, by calling the `Dispose` procedure.

note Memory management in general and the way the heap works in particular are covered in Chapter 13. In short, the heap is a (large) area of memory in which you can allocate and release blocks of memory in no given order. As an alternative to `New` and `Dispose` you can use `GetMem` and `FreeMem`, which require the developer to provide the size of the allocation (while the compiler determines the allocation size automatically in case of `New` and `Dispose`). In cases the allocation size is not known at compile time, `GetMem` and `FreeMem` become handy.

Here is a code snippet that allocates memory dynamically:

```
var
  P: ^Integer;
begin
  // initialization
  New (P);
  // operations
  P^ := 20;
  Show (P^.ToString);
  // termination
  Dispose (P);
```

170 - 05: Arrays and Records

If you don't dispose of the memory after using it, your program may eventually use up all the available memory and crash. The failure to release memory you don't need any more is known as a *memory leak*.

warning To be safer the code above should indeed use an exception handling `try-finally` block, a topic I decided not to introduce at this point of the book, but I'll cover later in Chapter 9.

If a pointer has no value, you can assign the `nil` value to it. You can test whether a pointer is `nil` to see if it currently refers to a value with a direct equality test or by using the specific `Assigned` function as shown below.

This kind of test is often used, because dereferencing (that is accessing the value at the memory address stored in the pointer) an invalid pointer causes a memory access violation (with slightly different effects depending on the operating system):

```
var
  P: ^Integer;
begin
  P := nil;
  Show (P^.ToString);
```

You can see an example of the effect of this code by running the `PointersTest` application project. The error you'll see (on Windows) should be similar to:

```
Access violation at address 0080B14E in module 'PointersTest.exe'. Read
of address 00000000.
```

One of the ways to make pointer data access safer, is to add a “pointer is not null” safe-check like the following:

```
if P <> nil then
  Show (P^.ToString);
```

As I mentioned earlier, an alternative way, which is generally preferable for readability reasons, is to use the `Assigned` pseudo-function:

```
if Assigned (P) then
  writeln (P^.ToString);
```

note `Assigned` is not a real function, because it is “resolved” by the compiler producing the proper code. Also, it can be used over a procedural type variable (or method reference) without actually invoking it, but only checking if it is assigned.

Object Pascal also defines a `Pointer` data type, which indicates untyped pointers (such as `void*` in the C language). If you use an untyped pointer you should use `GetMem` instead of `New` (indicating the number of bytes to allocate, given this value cannot be inferred from the type itself). The `GetMem` procedure is required each time the size of the memory variable to allocate is not defined.

The fact that pointers are seldom necessary in Object Pascal is an interesting advantage of this language. Still, having this feature available can help in implementing some extremely efficient low level functions and when calling the API of an operating system.

In any case, understanding pointers is important for advanced programming and for a full understanding of language object model, which uses pointers (generally called references) behind the scenes.

warning When a variable holds a pointer to a second variable and that second variable goes out of scope or is freed (if dynamically allocated) the pointer would just refer to memory location either undefined or holding some other data. This can lead to very hard to find bugs.

File Types, Anyone?

The last Object Pascal data type constructor covered (briefly) in this chapter is the *file* type. File types represent physical disk files, certainly a peculiarity of the original Pascal language, given very few old or modern programming languages include the notion of a file as a primitive data type. The Object Pascal language has a `file` keyword, which is a type specifier, like `array` or `record`. You use `file` to define a new type, and then you can use the new data type to declare new variables:

```
type
  IntFile = file of Integers;
var
  IntFile1: IntFile;
```

It is also possible to use the `file` keyword without indicating a data type, to specify an untyped file. Alternatively, you can use the `TextFile` type, defined in the `System` unit of the Run Time Library to declare files of ASCII characters (or, more correctly in these times, files of bytes).

Direct use of files, although still supported, is less and less common these days, as the Run Time Library includes many classes for managing binary and text files at a much higher level (including the support for Unicode encodings for text files, for example).

Delphi applications generally use the RTL streams (the `TStream` and derived classes) to handle any complex file read and write operations. Streams represent virtual files, which can be mapped to physical files, to a memory block, to a socket, or any other continuous series of bytes.

172 - 05: Arrays and Records

One area when you still see some of the old time file management routines in use is when you write console applications, where you can use `write`, `writeln`, `read`, and related function for operating with a special file, which is the standard input and standard output (C and C++ have similar support for input and output from the console, and many other languages offer similar services).

06: all about strings

Character strings are one of the most commonly used data types in any programming language. Object Pascal makes string handling fairly simple, yet very fast and extremely powerful. Even if the basics of strings are easy to grasp and I've used strings for output in the previous chapters, behind the scenes the situation is a little more complex than it might seem at first sight. Text manipulation involves several closely related topics worth exploring: to fully understand string processing you need to know about Unicode representations, understand how strings map to arrays of characters, and learn about some of the most relevant string operations of the run time library, including saving string to text files and loading them.

Object Pascal has several options for string manipulation and makes available different data types and approaches. The focus of the chapter will be on the standard string data type, but I'll also devote a little time to older string types like `AnsiString` that you can still use in Delphi compilers. Before we get to that, though, let me start from the beginning: the Unicode representation.

Unicode: An Alphabet for the Entire World

Object Pascal string management is centered around the Unicode character set and, particularly, the use of one of its representations, called UTF-16. Before we get to the technical details of the implementation, it is worth devoting a few sections to fully understanding the Unicode standard.

The idea behind Unicode (which is what makes it simple and complex at the same time) is that every single character in all known alphabets of the world has its own description, a graphical representation, and a unique numeric value (called a Unicode *code point*).

note The reference web site of the Unicode consortium is <http://www.unicode.org>, which a rich amount of documents. The ultimate reference is “The Unicode Standard” book, which can be found at <http://www.unicode.org/book/aboutbook.html>.

Not all developers are familiar with Unicode, and many still think of characters in terms of older, limited representations like ASCII and in terms of ISO encoding. By having a short section covering these older standards, you'd better appreciate the peculiarities (and the complexity) of Unicode.

Characters from the Past: from ASCII to ISO Encodings

Character representations started with the American Standard Code for Information Interchange (ASCII), which was developed in the early '60s as a standard encoding of computer characters, encompassing the 26 letters of the English alphabet, both lowercase and uppercase, the 10 numerical digits, common punctuation symbols, and a number of control characters (still in use today).

ASCII uses a 7 bit encoding system to represent 128 different characters. Only characters between #32 (Space) and #126 (Tilde) have a visual representation, as show in Figure 6.1 (extracted from an Object Pascal application running on Windows).

Figure 6.1:

A table with the printable ASCII character set

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0																
16																
32		!	"	#	\$	%	&	'	()	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{		}	~	

While ASCII was certainly a foundation (with its basic set of 128 characters that are still part of the core of Unicode), it was soon superseded by extended versions that used the 8th bit to add another 128 characters to the set.

Now the problem is that with so many languages around the world, there was no simple way to figure out which other characters to include in the set (at times indicated as ASCII-8). To make the story short, Windows adopted a different set of characters, called a *code page*, with a set of characters depending on your locale configuration and version of Windows. Beside Windows code pages there are many other standards based on a similar *paging* approach, and those pages became part of international ISO standards.

The most relevant was certainly the ISO 8859 standard, which defines several *regional* sets. The most used set (well, the one used in most Western countries to be a little more precise) is the Latin set, referenced as ISO 8859-1.

note Even if partially similar, Windows 1252 code page doesn't fully conform to the ISO 8859-1 set. Windows adds extra characters like the € symbol, extra quotation marks, and more, in the area from 128 to 150. Differently from all other values of the Latin set, those Windows extensions do not conform with the Unicode code points.

Unicode Code Points and Graphemes

If I really want to be precise, I should include one more concept beyond that of code points. At times, in fact, multiple code points could be used to represent a single *grapheme* (a visual character). This is generally not a letter, but a combination of letters or letters and symbols. For example, if you have a sequence of the code point representing the Latin letter *a* (#\$0061) followed by the code point representing the grave accent (#\$0300), this should be displayed as a single accented character.

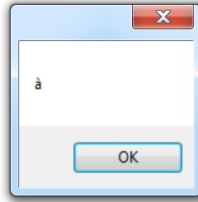
176 - 06: All About Strings

In Object Pascal coding terms, if you write the following (part of the `CodePoints` application project), the message will have one single accented character, as in Figure 6.2.

```
var
  Str: String;
begin
  Str := #$0061 + #$0300;
  ShowMessage (Str);
```

Figure 6.2:

A single grapheme can be the result of multiple code points



In this case we have two characters, representing two code points, but only one grapheme (or visual elements). The fact is that while in the Latin alphabet you can use a specific Unicode code point to represent the given grapheme (*letter a with grave accent is code point \$0061*), in other alphabets combining Unicode code points is the only way to obtain a given grapheme (and the correct output).

Even if the display is that of an accented character, there is no automatic normalization or transformation of the value (only a proper display), so the string internally remains different from one with the single character à.

note The rendering of graphemes from multiple code points might depend on specific support from the operating system and on text rendering techniques being used, so you might find out that for some of the graphemes not all operating systems offer the correct output.

From Code Points to Bytes (UTF)

While ASCII used a direct and easy mapping of character to their numeric representation, Unicode uses a more complex approach. As I mentioned, every element of the Unicode alphabet has an associated code point, but the mapping to the actual representation is often more complicated.

One of the elements of confusion behind Unicode is that there are multiple ways to represent the same code point (or Unicode character numerical value) in terms of actual storage, of physical bytes, in memory or on a file.

The issue stems from the fact that the only way to represent all Unicode code points in a simple and uniform way is to use four bytes for each code point. This accounts for a fixed-length representation (each character requires always the same amount of bytes), but most developers would perceive this as too expensive in memory and processing terms.

note In Object Pascal the Unicode Code Points can be represented directly in a 4-bytes representation by using the `UCS4Char` data type.

That's why the Unicode standard defines other representations, generally requiring less memory, but in which the number of bytes for each symbol is different, depending its code point. The idea is to use a smaller representation for the most common elements, and a longer one for those infrequently encountered.

The different physical representations of the Unicode code points are called Unicode Transformation Formats (or UTF). These are algorithmic *mappings*, part of the Unicode standard, that map each code point (the absolute numeric representation of a character) to a unique sequence of bytes representing the given character. Notice that the mappings can be used in both directions, converting back and forth between different representations.

The standard defines three of these formats, depending on how many bits are used to represent the initial part of the set (the initial 128 characters): 8, 16, or 32. It is interesting to notice that all three forms of encodings need at most 4 bytes of data for each code point.

- **UTF-8** transforms characters into a variable-length encoding of 1 to 4 bytes. UTF-8 is popular for HTML and similar protocols, because it is quite compact when most characters (like tags in HTML) fall within the ASCII subset.
- **UTF-16** is popular in many operating systems (including Windows and macOS) and development environments. It is quite convenient as most characters fit in two bytes, reasonably compact, and fast to process.
- **UTF-32** makes a lot of sense for processing (all code points have the same length), but it is memory consuming and has limited use in practice.

There is a common misconception that UTF-16 can map directly all code points with two bytes, but since Unicode defines over 100,000 code points you can easily figure out they won't fit into 64K elements. It is true, however, that at times developers use only a subset of Unicode, to make it fit in a 2-bytes-per-character fixed-length representation. In the early days, this subset of Unicode was called UCS-2, now you often see it referenced as Basic Multilingual Plane (BMP). However, this is only a subset of Unicode (one of many *planes*).

note A problem relating to multi-byte representations (UTF-16 and UTF-32) is which of the bytes comes first? According to the standard, all forms are allowed, so you can have a UTF-16 BE (big-endian) or LE (little-endian), and the same for UTF-32. The big-endian byte serialization has the most significant byte first, the little-endian byte serialization has the least significant byte first. The bytes serialization is often marked in files along with the UTF representation with a header called Byte Order Mark (BOM).

The Byte Order Mark

When you have a text file storing Unicode characters, there is a way to indicate which is the UTF format being used for the code points. The information is stored in a header or marker at the beginning of the file, called Byte Order Mark (BOM). This is a signature indicating the Unicode format being used and the byte order form (little or big endian – LE or BE). The following table provides a summary of the various BOMs, which can be 2, 3, or 4 bytes long:

00 00 FE FF	UTF-32, big-endian
FF FE 00 00	UTF-32, little-endian
FE FF	UTF-16, big-endian
FF FE	UTF-16, little-endian
EF BB BF	UTF-8

We'll see later in this chapter how Object Pascal manages the BOM within its streaming classes. The BOM appears at the very beginning of a file with the actual Unicode data immediately following it. So a UTF-8 file with the content *AB* contains five hexadecimal values (3 for the BOM, 2 for the letters):

■ EF BB BF 41 42

If a text file has none of these signatures, it is generally considered as an ASCII text file, but it might as well contain text with any encoding.

note On the other hand, when you are receiving data from a web request or through other Internet protocols, you might have a specific header (part of the protocol) indicating the encoding, rather than relying on a BOM.

Looking at Unicode

How do we create a table of Unicode characters like those I displayed earlier for ASCII ones? We can start by displaying code points in the Basic Multilingual Plane (BMP), excluding what are called surrogate pairs.

note Not all numeric values are true UTF-16 code points, since there are some non-valid numerical values for characters (called surrogates) used to form a paired code and represent code points above 65535. A good example of a surrogate pair is the symbol used in music scores for the F (or bass) clef. It's code point 1D122 which is represented in UTF-16 by two values, D834 followed by DD22.

Displaying all of the elements of the BMP would require a $256 * 256$ grid, hard to accommodate on screen. This is why the `ShowUnicode` application project has a tab with two pages: The first tab has the primary selector with 256 blocks, while the second page shows a grid with the actual Unicode elements, one section at a time. This program has a little more of a user interface than most others in the book, and you can simply skim through its code if you are only interested in its output (and not the internals).

When the program starts, it fills the `ListView` control in the first page of the `TabControl` with 256 entries, each indicating the first and last character of a group of 256. Here is the actual code of the `oncreate` event handler of the form and a simple function used to display each element, producing the output of Figure 6.3:

```
// helper function
function GetCharDescr (nChar: Integer): string;
begin
    if Char(nChar).IsControl then
        Result := 'Char #' + IntToStr (nChar) + ' [ ]'
    else
        Result := 'Char #' + IntToStr (nChar) +
            ' [' + Char (nChar) + ']';
end;

procedure TForm2.FormCreate(Sender: TObject);
var
    I: Integer;
    ListItem: TListItem;
begin
    for I := 0 to 255 do // 256 pages * 256 characters each
    begin
        ListItem := ListView1.Items.Add;
        ListItem.Tag := I;
        if (I < 216) or (I > 223) then
            ListItem.Text :=
                GetCharDescr(I*256) + '/' + GetCharDescr(I*256+255)
        else
            ListItem.Text := 'Surrogate Code Points';
```

180 - 06: All About Strings

```
end;  
end;
```

Figure 6.3:

The first page of the ShowUnicode application project has a long list of sections of Unicode characters



Notice how the code saves the number of the “page” in the `Tag` property of the items of the `ListView`, an information used later to fill a page. As a user selects one of the items, the application moves to the second page of the `TabControl`, filling its string grid with the 256 characters of the section:

```
procedure TForm2.ListViewItemClick(const Sender: TObject;  
    const AItem: TListViewItem);  
var  
    I, NStart: Integer;  
begin  
    NStart := AItem.Tag * 256;  
    for I := 0 to 255 do  
        begin  
            StringGrid1.Cells[I mod 16, I div 16] :=  
                IfThen(not Char(I + NStart).IsControl, Char(I + NStart), '');  
        end;  
    TabControl1.ActiveTab := TabItem2;
```

The `IfThen` function used in the code above is a two way test: If the condition passed in the first parameter is true, the function returns the value of the second parameter; if not, it returns the value of the third one. The test in the first parameter

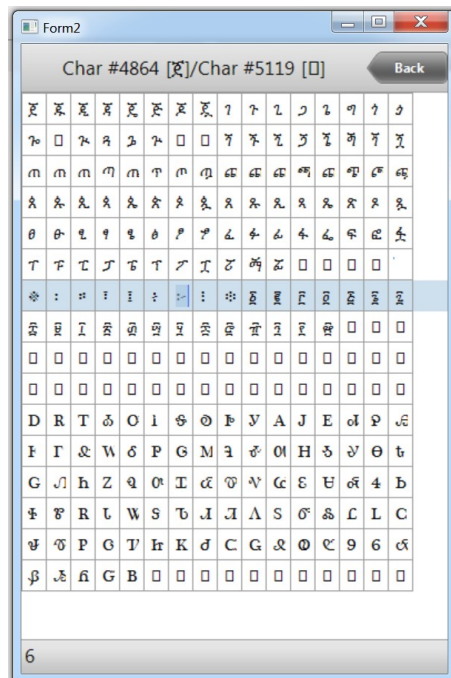
uses the `IsControl` method of the `Char` type helper, to filter out non-printable control characters.

note The `IfThen` function operates more or less like the `?:` operator of most programming languages based on the C syntax. There is a version for strings and a separate one for Integers. For the string version you have to include the `System.StrUtils` unit, for the Integer version of `IfThen` the `System.SysUtils` unit.

The grid of Unicode characters produced by the application is visible in Figure 6.4. Notice that the output varies depending on the ability of the selected font and the specific operating system to display a given Unicode character.

Figure 6.4:

The second page of the `ShowUnicode` application project has some of the actual Unicode characters



The Char Type Revisited

After this introduction to Unicode, let's get back to the real topic of this chapter, which is how the Object Pascal language manages characters and strings. I introduced the Char data type in Chapter 2, and mentioned some of the type helper functions available in the Character unit. Now that you have a better understanding of Unicode, it is worth revisiting that section and going through some more details.

First of all, the Char type does not invariably represent a Unicode code point. The data type, in fact, uses 2 bytes for each element. While it does represent a code point for elements in Unicode's Basic Multi-language Plane (BMP), a Char can also be part of a pair of surrogate values, representing a code point.

Technically, there is a different type you could use to represent any Unicode code point directly, and this is the UCS4Char type, which used 4 bytes to represent a value). This type is rarely used, as the extra memory required is generally hard to justify, but you can see that the Character unit (covered next) also includes several operations for this data type.

Back to the Char type, remember it is an ordinal type (even if a rather large one), so it has the notion of sequence and offers code operations like `Ord`, `Inc`, `Dec`, `High`, and `Low`. Most extended operations, including the specific type helper, are not part of the basic system RTL units but require the inclusion of the Character unit.

Unicode Operations With The Character Unit

Most of the specific operations for Unicode characters (and also Unicode strings, of course) are defined in a special unit called `System.Character`. This unit defines the `TCharHelper` helper for the Char type, which lets you apply operations directly to variables of that type.

note The Character unit also defines a `TCharacter` record, which is basically a collection of static class functions, plus a number of global routines mapped to these methods. These are older, deprecated functions, given that now the preferred way to work on the Char type at the Unicode level is the use of the class helper.

The unit also defines two interesting enumerated types. The first is called `TUnicodeCategory` and maps the various characters in broad categories like control, space, uppercase or lowercase letter, decimal number, punctuation, math symbol, and many more. The second enumeration is called `TUnicodeBreak` and defines the fam-

ily of the various spaces, hyphen, and breaks. If you are used to ASCII operations, this is a big change.

Numbers in Unicode are not only the characters between 0 and 9; spaces are not limited to the character #32; and so on for many other assumption of the (much simpler) 256-elements alphabet.

The Char type helper has over 40 methods that comprise many different tests and operations. They can be used for:

- Getting the numeric representation of the character (`GetNumericValue`).
- Asking for the category (`GetUnicodeCategory`) or checking it against one of the various categories (`IsLetterOrDigit`, `IsLetter`, `IsDigit`, `IsNumber`, `IsControl`, `IsWhiteSpace`, `IsPunctuation`, `IsSymbol`, and `IsSeparator`). I used the `IsControl` operation in the previous demo.
- Checking if it is lowercase or uppercase (`IsLower` and `IsUpper`) or converting it (`ToLower` and `ToUpper`).
- Verifying if it is part of a UTF-16 surrogate pair (`IsSurrogate`, `IsLowSurrogate`, and `IsHighSurrogate`) and convert surrogate pairs in various ways.
- Converting it to and from UTF32 (`ConvertFromUtf32` and `ConvertToUtf32`) and UCS4Char type (`ToUCS4Char`).
- Checking if it is part of a given list of characters (`IsInArray`).

Notice that some of these operations can be applied to the type as a whole, rather than to a specific variable. In that can you have to call them using the Char type as prefix, as in the second code snippet below.

To experiment a bit with these operations on Unicode characters, I've create an application project called `CharTest`. One of the examples of this demo is the effect of calling uppercase and lowercase operations on Unicode elements. In fact, the classic `UpCase` function of the RTL works only for the base 26 English language characters of the ANSI representation, while it fails some Unicode character that do have a specific uppercase representations (not all alphabets have the concept of uppercase, so this is not a universal notion).

To test this scenario, in the `CharTest` application project I've added the following snippet that tries to convert an accented letter to uppercase:

```
var
  Ch1: Char;
begin
  Ch1 := 'ù';
  Show ('UpCase ù: ' + UpCase(Ch1));
  Show ('ToUpper ù: ' + Ch1.ToUpper);
```


184 - 06: All About Strings

The traditional `upcase` call won't convert the Latin accented character, while the `ToUpper` function works properly:

```
UpCase ù: ù  
ToUpper ù: Ù
```

There are many Unicode-related features in the `Char` type helper, like those highlighted in the code below, which defines a string as including also a character outside of the BMP (the first 64K of Unicode code points). The code snippet, also part of the `CharTest` application project, has a few tests on the various elements of the string, all returning `True`:

```
var  
  Str1: string;  
begin  
  Str1 := '1.' + #9 + Char.ConvertFromUtf32 (128) +  
    Char.ConvertFromUtf32($1D11E);  
  ShowBool (Str1.Chars[0].IsNumber);  
  ShowBool (Str1.Chars[1].IsPunctuation);  
  ShowBool (Str1.Chars[2].IsWhiteSpace);  
  ShowBool (Str1.Chars[3].IsControl);  
  ShowBool (Str1.Chars[4].IsSurrogate);  
end;
```

The display function used in this case is an adapted version:

```
procedure TForm1.ShowBool(value: Boolean);  
begin  
  Show(BoolToStr (Value, True));  
end;
```

note Unicode code point \$1D11E is *musical symbol G clef*.

Unicode Character Literals

We have seen in several examples that you can assign an individual character literal or a string literal to a variable of the string type. In general using the numeric representation of a character with the `#` prefix is quite simple. There are some exceptions, though. For backwards compatibility, plain character literals are converted depending on their context. Take the following simple assignment of the numerical value 128, which probably indicates the use of the Euro currency symbol (€):

```
var  
  Str1: string;  
begin  
  Str1 := #$80;
```

This code is not Unicode compliant, as the code point for that symbol is 8364. The value, in fact, doesn't come from the official ISO codepages, but it was a specific

Microsoft implementation for Windows. To make it easier to move existing code to Unicode, the Object Pascal compiler can treat 2-digit string literals as ANSI characters (which might depend on your actual code page). Surprisingly enough if you take that value, convert it to a Char, and display it again... the numerical representation will change to the correct one. So by executing the statement:

```
┌ Show (Str1 + ' - ' + IntToStr (Ord (Str1[1])));
```

I'll get the output:

```
┌ € - 8364
```

Given you might prefer fully migrating your code and getting rid of older ANSI-based literal values, you can change the compiler behavior by using the special directive `$HIGHCHARUNICODE`. This directive determines how literal values between `#$80` and `#$FF` are treated by the compiler. What I discussed earlier is the effect of the default option (OFF). If you turn it on, the same program will produce this output:

```
┌ - 128
```

The number is interpreted as an actual Unicode code point and the output will contain a non-printable control character. Another option to express that specific code point (or any Unicode code point below `#$FFFF`) is to use the four-digits notation:

```
┌ str1 := #$0080;
```

This is not interpreted as the Euro currency symbol regardless of the setting of the `$HIGHCHARUNICODE` directive.

note The code above and the matching demo work only for a US or Western European locale. With other locales the characters between 128 and 255 are interpreted differently.

What is nice is that you can use the four digits notation to express far eastern characters, like the following two Japanese characters:

```
┌ str1 := #$3042#$3044;  
  Show (Str1 + ' - ' + IntToStr (Ord (Str1.Chars[0])) +  
        ' - ' + IntToStr (Ord (Str1.Chars[1])));
```

displayed as (along with their Integer representation):

```
┌ あい - 12354 - 12356
```

You can also use literal elements over `#$FFFF` that will be converted to the proper surrogate pair.

What about 1-Byte Chars?

As I mentioned earlier, the Object Pascal language maps the `Char` type to `wideChar`, but it still defines the `AnsiChar` type, mostly for compatibility with existing code. The general recommendation is to use the `Byte` type for a one-byte data structure, but it is true that `AnsiChar` can be handy when doing 1-byte characters processing.

While for several versions `AnsiChar` wasn't available on mobile platforms, starting with 10.4 this data type works the same across all Delphi compilers. When mapping data to a platform API or when saving to files, you should generally stay away from the old one-byte chars, even if supported. Using a Unicode encoding is by far the preferred approach. It is true, though, that 1-byte characters processing can be faster and use less memory than the 2-bytes counterpart.

The String Data Type

The string data type in Object Pascal is way more sophisticated than a simple array of characters, and has features that go well beyond what most programming languages do with similar data types. In this section I'll introduce the key concepts behind this data type, and in coming sections we'll explore some of these features in more details.

In the following bullet list I've captured the key concepts for understanding how strings work in the language (remember, you can use string without knowing much of this, as the internal behavior is very transparent):

- Data for the string type is ***dynamically allocated*** on the heap. A string variable is just a reference to the actual data. Not that you have to worry much about this, as the compiler handles this transparently. Like for a dynamic array, as you declare a new string, this is empty.
- While you can assign data to a string in many ways, you can also ***allocate a specific memory area*** of a given size by calling the `SetLength` function. The parameter is the number of characters (of 2 bytes each), the string should be able to have. When you extend a string, the existing data is preserved (but it might be moved to a new physical memory location). When you reduce the size, some of the content will likely be lost. Setting the length of a string is seldom necessary. The only common case is when you need to pass a string buffer to an operating system function for the given platform.

- If you want to **increase the size** of a string in memory (by concatenating it with another string) but there is something else in the adjacent memory, then the string cannot grow in the same memory location, and a full copy of the string must therefore be made in another location.
- To clear a string you don't operate on the reference itself, but can simply set it to an empty string, that is `''`. Or you can use the `Empty` constant, which corresponds to that value.
- According to the rules of Object Pascal, the **length of a string** (which you can obtain by calling `Length`) is the number of valid elements, not the number of allocated elements. Differently from C, which has the concept of a string terminator (`#0`), all versions of Pascal since the early days tend to favor the use of a specific memory area (part of the string) where the actual length information is stored. At times, however, you'll find strings that also have the terminator.
- Object Pascal strings use a **reference-counting** mechanism, which keeps track of how many string variables are referring to a given string in memory. Reference counting will free the memory when a string isn't used anymore—that is, when there are no more string variables referring to the data... and the reference count reaches zero.
- Strings use a **copy-on-write** technique, which is highly efficient. When you assign a string to another or pass one to a string parameter, no data is copied and the reference count is increased. However, if you do change the content of one of the references, the system will first make a copy and then alter only that copy, with the other references remaining unchanged.
- The use of **string concatenation** for adding content to an existing string is generally very fast and has no significant drawback. While there are alternative approaches, concatenating strings is fast and powerful. This is not true for many programming languages these days.

Now I can guess this description can be a little confusing, so let's look at the use of strings in practice. In a while I'll get to a demo showcasing some of the operations above, including reference counting and copy-on-write. Before we do so, however, let me get back to the string helper operations and some other fundamental RTL functions for strings management.

Before we proceed further, let me examine some of the elements of the previous list in terms of actual code. Given string operations are quite seamless it is difficult to fully grasp what happens, unless you start looking inside the strings memory structure, which I'll do later in this chapter, as it would be too advanced for now. So let's start with some simple string operations, extracted from the `Strings101` application project:

188 - 06: All About Strings

```
var
  String1, String2: string;
begin
  String1 := 'hello world';
  String2 := String1;
  Show ('1: ' + String1);
  Show ('2: ' + String2);
  String2 := String2 + ', again';
  Show ('1: ' + String1);
  Show ('2: ' + String2);
end;
```

This first snippet, when executed, shows that if you assign two strings to the same content, modifying one won't affect the other. That is, `String1` is not affected by the changes to `String2`:

```
1: hello world
2: hello world
1: hello world
2: hello world, again
```

Still, as we'll figure out better in a later demo, the initial assignment doesn't cause a full copy of the string, the copy is delayed (again, a feature called *copy-on-write*).

Another important feature to understand is how the length is managed. If you ask for the length of a string, you get the actual value (which is stored in the string meta data, making the operation very fast). But if you call `SetLength`, you are allocating memory, which most often will be not initialized. This is generally used when passing the string as a buffer to an external system function. If you need a blank string, instead, you can use the pseudo-constructor (`Create`). Finally, you can use `SetLength` to trim a string. All of these are demonstrated by the following code:

```
var
  String1: string;
begin
  String1 := 'hello world';
  Show (String1);
  Show ('Length: ' + String1.Length.ToString);

  SetLength (String1, 100);
  Show (String1);
  Show ('Length: ' + String1.Length.ToString);

  String1 := 'hello world';
  Show (String1);
  Show ('Length: ' + String1.Length.ToString);

  String1 := string1 + String.Create(' ', 100);
  SetLength (String1, 100);
  Show (String1);
  Show ('Length: ' + String1.Length.ToString);
```

The output is more or less the following:

[illegible]

The third concept I want to underline in this section is that of an empty string. A string is empty when its content is an empty string. For both assignment and testing you can use two consecutive quotes, or specific functions:

```
var
  String1: string;
begin
  String1 := 'hello world';
  if String1 = '' then
    Show('Empty')
  else
    Show('Not empty');

  String1 := ''; // or String1.Empty;
  if String1.IsEmpty then
    Show('Empty')
  else
    Show('Not empty');
```

With this simple output:

Not empty
Empty

Passing Strings as Parameters

As I've explained, if you assign a string to another, you are just copying a reference, while the actual string in memory is not duplicated. However, if you write code that changes that string, the string is first copied (only at that point), and then modified.

Something very similar happens when you pass a string as parameter to a function or procedure. By default, you get a new reference and if you modify the string in the function, the change doesn't affect the original string. If you want a different behavior, that is the ability to modify the original string in the function, you need to pass the string by reference, using the `var` keyword (as it happens for most other simple and managed data types).

But what if you don't modify the string passed as parameter? In that case, you can apply an actual optimization by using the `const` modifier for the parameter. In this

190 - 06: All About Strings

case the compiler won't let you change the string in the function or procedure, but it will also optimize the parameter passing operation. In fact, a `const` string doesn't require the function to increase the string reference count when it starts and decrease it when it ends. While these operations are very fast, executing those thousands or millions of times will add a slight overhead to your programs. This is why passing string as `const` is recommended in cases where the function doesn't have to modify the value of the string parameter (although there are potential issues covered in the note below).

In coding terms, these are the declarations of three procedures with a string parameters passed in different ways:

```
procedure ShowMsg1 (Str: string);  
procedure ShowMsg2 (var Str: string);  
procedure ShowMsg3 (const Str: string);
```

note In recent years, there has been a strong push for passing all string parameters as `const`, unless the functions and methods modify these strings. There is a very big caveat, though. For a constant string parameter, the compiler takes the string reference and doesn't "manage" it (no reference counting, etc), treating it like a pointer to the memory location. The compiler properly checks that the code of the routine doesn't change the string parameter. However it has no control on what happens to the original string the parameter refers to. Changes to that string can affect its memory layout and location, something a regular string parameter can handle (strings with multiple references do a copy-on-write operation automatically) while a constant string parameter will ignore. In other words, a change to the original string makes the `const` parameter referring to it invalid, and its use will most likely cause a memory access error.

The Use of [] and String Characters Counting Modes

As you are likely to know if you have used Object Pascal or any other programming language, a key string operation is accessing one of the elements of a string, something often achieved using the square brackets notation (`[]`), in the same way you access the elements of an array.

In Object Pascal there are two slightly different ways to perform these operations:

- The `chars[]` string type helper operation (the entire list is in the next section) is a read only character access that uses a 0-based index.
- The standard `[]` string operator supports both reading and writing, and uses by default the classic Pascal one-based index. This setting can be modified with a compiler directive.

I'm going to provide some clarifications on this matter after a short historical note below. The reason for this note, which you can skip if not interested, is that it would be difficult to understand why the language behaves in the current way without looking at what happened over time.

note Let me look back in time for a second, to explain you how we got here today. In the early days of the Pascal language, strings were treated like an array of characters in which the first element (that is the element zero of the array) was used to store the number of valid characters in the string, or the string length. In those days, while the C language had to recompute the length of a string each time, looking for a NULL terminator, Pascal code could just make a direct check to that byte. Given that the byte number zero was used for the length, it happened that the first actual character stored in the string was at position one.

Over time, almost all other languages had zero-based strings and arrays. Later, Object Pascal adopted zero-based dynamic arrays and most of the RTL and component libraries used zero-based data structures, with strings being a significant exception.

When moving to the mobile world, the Object Pascal language designers decided to give “priority” to zero-based strings, allowing developers to still use the older model in case they had existing Object Pascal source code to move over, controlling the behavior with a compiler directive. In Delphi 10.4, though, the original decision was reverted to account for more consistency of the source code regardless of the target platform. In other words, it was decided to favor the “single source multiplatform” goal over the “be like other modern languages” goal.

If we want to draw a comparison to better explain the differences in the base of the index, consider how floors are counted in Europe and in North America (I honestly don't know about the rest of the world). In Europe the ground floor is floor 0, and the first floor is the one above it (at times formally indicated as “floor one above ground”). In North America, the first floor is the ground floor, and the second first is the first above ground level. In other words, America uses a 1-based floor index, while Europe uses a 0-based floor index.

For strings, the largest majority of programming languages uses the 0-based notation, regardless of the continent they were invented. Delphi and most Pascal dialects use the 1-based notation.

Let me explain the situation with string indexes a little better. As I mentioned above, the `Chars[]` notation invariably uses a zero-based index. So if you write

```
var
  String1: string;
begin
  String1 := 'hello world';
  Show (String1.Chars[1]);
```

the output will be:

```
l e
```

What if you use the direct `[]` notation, that is what will be the output of:

192 - 06: All About Strings

```
| Show (String1[1]);
```

By default the output will be `h`. However it might be `e` if the compiler define `$ZEROBASEDSTRING` has been turned ON. The recommendation at this time (that is, after the Delphi 10.4. release) is to move to one-based strings in all of your code and avoid tackling with different models.

But what if you want to write code that works regardless of the `$ZEROBASEDSTRING` setting? You can abstracting the index, for example using `Low(string)` as the index of the first value and `High(string)` for the last value. These work returning the proper value depending on the local compiler setting for the string base:

```
var
  S: string;
  I: Integer;
begin
  S := 'Hello world';
  for I := Low (S) to High (S) do
    Show(S[I]);
```

In other words, a string *invariably* has elements ranging from the result of the `Low` function to that of the `High` function applied to the same string.

note A string is just a string, and the *concept* of a zero-based string is completely wrong. The data structure in memory is not different in any way, so you can pass any string to any function that uses a notation with any base value, and there is no problem at all. In other words, if you have a code fragment accessing to strings with a zero-based notation you can pass the string to a function that is compiled using the settings for a one-based notation.

Concatenating Strings

I have already mentioned that unlike other languages, Object Pascal has full support for direct string concatenation, which is actually a rather fast operation. In this chapter I'll just show you some string concatenations code, while doing some speed testing. Later on, in Chapter 18, I'll briefly cover the `TStringBuilder` class, which follows the .NET notation for assembling a string out of different fragments. While there are reasons to use `TStringBuilder`, performance is not the most relevant one (as the following example will show).

So, how do we concatenate strings in Object Pascal? Simply by using the `+` operator:

```
var
  Str1, Str2: string;
begin
  Str1 := 'Hello, ';
  Str2 := ' world';
  Str1 := Str1 + Str2;
```

Notice how I used the `str1` variable both on the left and on the right of the assignment, adding some more content to an existing string rather than assigning to a brand new one. Both operations are possible, but adding content to an existing string is where you can get some nice performance.

This type of concatenation can be also done in a loop, like the following extracted from the `LargeString` application project:

```

uses
  System.Diagnostics;

const
  MaxLoop = 2000000; // two million

var
  Str1, Str2: string;
  I: Integer;
  T1: TStopwatch;
begin
  Str1 := 'Marco ';
  Str2 := 'Cantu ';

  T1 := TStopwatch.StartNew;
  for I := 1 to MaxLoop do
    Str1 := Str1 + Str2;

  T1.Stop;
  Show('Length: ' + Str1.Length.ToString);
  Show('Concatenation: ' + T1.ElapsedMilliseconds.ToString);
end;

```

By running this code, I get the following timing on a Windows virtual machines and on an Android device (the computer is quite a bit faster):

```

Length: 12000006           // windows (in a VM)
Concatenation: 59

Length: 12000006           // Android (on device)
Concatenation: 991

```

The application project has also similar code based on the `TStringBuilder` class. While I don't want to get to the details of that code (again, I'll describe the class in Chapter 18) I want to share the actual timing, for comparison with the plain concatenation timing just displayed

```

Length: 12000006           // windows (in a VM)
StringBuilder: 79

Length: 12000006           // Android (on device)
StringBuilder: 1057

```

As you can see, concatenation can be safely considered the fastest option.

The String Helper Operations

Given the importance of the string type, it should come to no surprise that the helper for this type has a rather long list of operations you can perform. And given its importance and the commonality of these operations in most applications, I think it is worth going through this list with some care.

There is a key difference between using the Delphi classic global string manipulation functions and the string helper methods: the classic operations assume one-based strings while the string helper operation use a zero-based logic!

I've logically grouped the string helper operations (most of which have many overloaded versions), shortly describing what they do, considering that quite often that their names are rather intuitive:

- Copy or partial copy operations like `Copy`, `CopyTo`, `Join`, and `SubString`
- String modification operations like `Insert`, `Remove`, and `Replace`
- For conversion from various data types to string, you can use `Parse` and `Format`
- Conversion to various data types, when possible can be achieved using `ToBoolean`, `ToInteger`, `ToSingle`, `ToDouble`, and `ToExtended` while you can turn a string into an array of characters with `ToCharArray`
- Fill a string white space or specific characters with `PadLeft`, `PadRight`, and one of the overloaded versions of `Create`. At the opposite, you can remove white space at one end of the string or both using `TrimRight`, `TrimLeft`, and `Trim`
- String comparison and equality test (`Compare`, `CompareOrdinal`, `CompareText`, `CompareTo`, and `Equals`)—but keep in mind you can also, to some extent, use the equality operator and the comparison operators
- Changing case with `LowerCase` and `UpperCase`, `ToLower` and `ToUpper`, and `ToUpperInvariant`
- Test the string content with operations like `Contains`, `StartsWith`, `EndsWith`. Search in the string can be done using `IndexOf` for finding the position of a given character (from the beginning or from a given location), the similar `IndexOfAny` (which looks for one of the elements of an array of characters), the `LastIndexOf` and `LastIndexOfAny` operations which work backwards from the end of the string, and the special purpose operations `IsDelimiter` and `LastDelimiter`
- Access general information about the string with functions like `Length`, which returns the number of characters, `CountChars`, which also takes surrogate pairs into account, `GetHashCode`, which returns a hash of the string, and the various tests for “emptiness” which include `IsEmpty`, `IsNullOrEmpty`, and `IsNullOrWhiteSpace`

- String special operations like `split`, which breaks a string into multiple ones based on a specific character, and removing or adding quotes around the string with `QuotedString` and `DeQuoted`
- And, finally, access individual characters with `Chars[]`, which has the numerical index of the element of the string among square brackets. This can be used only for reading a value (not for changing it) and uses a zero-based index like all other string helper operations.

It is important to notice, in fact, that all of the string helper methods have been build following the string convention used by many other languages, which includes the concept that string elements start with zero and go up to the length of the string minus one. In other words, as I already mentioned earlier but is worth underlining again, all string helper operations use zero-based indexes as parameters and return values.

note The `Split` operation is relatively new to the Object Pascal RTL. A previously common approach was loading a string in a string list, after setting a specific line separator, and later access the individual strings, or lines. The `Split` operation is significantly more efficient and flexible.

Given the large amount of operations you can apply directly to strings, I could have created several projects demonstrating these capabilities. Instead, I'll stick to a few relatively simple operations, albeit very common ones.

The `StringHelperTest` application project has two buttons. In each of them the first part of the code builds and displays a string:

```
var
  Str1, Str2: string;
  I, NIndex: Integer;
begin
  Str1 := '';

  // create string
  for I := 1 to 10 do
    Str1 := Str1 + 'Object ';

  Str2:= string.Copy (Str1);
  Str1 := Str2 + 'Pascal ' + Str2.Substring (10, 30);
  Show(Str1);
```

Noticed how I used the `Copy` function, to create a unique copy of the data of the string, rather than an alias... even if in this particular demo it won't have made any difference. The `Substring` call at the end is used to extract a portion of the string. The resulting text is:

```
Object Object Object Object Object Object Object Object Object Object
Pascal ect Object Object Object Objec
```

196 - 06: All About Strings

After this initialization, the first button has code for searching for a substring and for repeating such a search, with a different initial index, to count the occurrences of a given string (in the example a single character):

```
// find substring
Show('Pascal at: ' +
    Str1.IndexOf ('Pascal').ToString);

// count occurrences
I := -1;
NCount := 0;
repeat
    I := Str1.IndexOf('o', I + 1); // search from next element
    if I >= 0 then
        Inc (NCount); // found one
until I < 0;

Show('o found: ' +
    NCount.ToString + ' times');
```

I know the repeat loop is not the simplest one: it starts with a negative index, as any following search begins with the index after the current one; it counts occurrences; and its termination is based on the fact that if the element is not found it returns -1. The output of the code is:

```
Pascal at: 70
o found: 14 times
```

The second button has code to perform a search and replace one or more elements of a string with something else. In the first part, it creates a new string copying the initial and final part and adding some new text in the middle. In the second, it uses the `Replace` function that can operate on multiple occurrences simply by passing to it the proper flag (`rfReplaceAll`).

This is the code:

```
// single replace
NIndex := Str1.IndexOf ('Pascal');
Str1 := Str1.Substring(0, NIndex) + 'object' +
    Str1.Substring(NIndex + ('Pascal').Length);
Show (Str1);

// multi-replace
Str1 := Str1.Replace('o', 'o', [rfReplaceAll]);
Show (Str1);
```

As the output is rather long and not easy to read, here I've listed only the central portion of each string:

```
...Object Pascal ect Object Object...
...Object Object ect Object Object...
...object object ect object object...
```

Again, this is just a minimal sampler of the rich string operations you can perform using the operations available for the string type using the string type helper.

More String RTL Functions

An effect of the decision to implement the string helper following the names of operations common in other programming languages is the fact that the names of the type operations often diverge from the traditional Object Pascal ones (which are still available as global functions today).

The following table has some of the *not-matching* functions names:

<i>global</i>	<i>string type helper</i>
Pos	IndexOf
IntToStr	Parse
StrToInt	ToInteger
CharOf	Create
StringReplace	Replace

note Remember that there is a big difference between the global and the Char helper operations: The first group uses a one-based notation for indexing elements within a string, while the latter group uses a zero-based notation (as explained earlier).

These are only the most commonly used functions of the string RTL that have changed name, while many others still use the same like `UpperCase` or `QuotedString`. The `System.SysUtils` unit has a lot more, and the specific `System.StrUtils` unit has also many functions focused on string manipulation that are not part of the string helper.

Some notable functions part of the `System.StrUtils` unit are:

- `ResemblesText`, which implements a *Soundex* algorithm (finding words with similar sound even if a different spelling);
- `DupeString`, which returns the requested number of copies of the given string;
- `IfThen`, which returns the first string passed if a condition is true, else it will return the second string (I used this function in a code snippet earlier in this chapter);
- `ReverseString`, which returns a string with the opposite characters sequence.

Formatting Strings

While concatenating string with the plus (+) operator and using some of the conversion functions you can indeed build complex strings out of existing values of various data types, there is a different and more powerful approach to formatting numbers, currency values, and other strings into a final string. Complex string formatting can be achieved by calling the `Format` function, a very traditional but still extremely common mechanism, not only in Object Pascal but in most programming languages.

history The family of “print format string” or `printf` functions date back to the early days of programming and languages like FORTRAN 66, PL/1, and ALGOL 68. The specific format string structure still in use today (and used by Object Pascal) is close to the C language `printf` function. For a historical overview you can refer to en.wikipedia.org/wiki/Printf_format_string.

The `Format` function requires as parameters a string with the basic text and some placeholders (marked by the % symbol) and an array of values, generally one for each of the placeholders. For example, to format two numbers into a string you can write:

```
| Format ('First %d, Second %d', [n1, n2]);
```

where `n1` and `n2` are two Integer values. The first placeholder is replaced by the first value, the second matches the second, and so on. If the output type of the placeholder (indicated by the letter after the % symbol) doesn't match the type of the corresponding parameter, a runtime error occurs. Having no compile-time type checking is actually the biggest drawback of using the `Format` function. Similarly, not passing enough parameters causes a runtime error.

The `Format` function uses an open-array parameter (a parameter that can have an arbitrary number of values or arbitrary data types, as covered in Chapter 5). Besides using %d, you can use one of many other placeholders defined by this function and briefly listed the following table. These placeholders provide a default output for the given data type. However, you can use further format specifiers to alter the default output. A width specifier, for example, determines a fixed number of characters in the output, while a precision specifier indicates the number of decimal digits. For example,

```
| Format ('%8d', [n1]);
```

converts the number `n1` into an eight-character string, right-aligning the text (use the minus (-) symbol to specify left-justification) filling it with white spaces. Here is the list of formatting placeholders for the various data types:

d (decimal)	The corresponding integer value is converted to a string of decimal digits.
x (hexadecimal)	The corresponding integer value is converted to a string of hexadecimal digits.
p (pointer)	The corresponding pointer value is converted to a string expressed with hexadecimal digits.
s (string)	The corresponding string, character, or PChar (pointer to a character array) value is copied to the output string.
e (exponential)	The corresponding floating-point value is converted to a string based on scientific notation.
f (floating point)	The corresponding floating-point value is converted to a string based on floating point notation.
g (general)	The corresponding floating-point value is converted to the shortest possible decimal string using either floating-point or exponential notation.
n (number)	The corresponding floating-point value is converted to a floating-point string but also uses thousands separators.
m (money)	The corresponding floating-point value is converted to a string representing a currency amount. The conversion is generally based on regional settings.

The best way to see examples of these conversions is to experiment with format strings yourself. To make this easier I've written the `FormatString` application project, which allows a user to provide formatting strings for a few predefined integer values.

The form of the program has an edit box above the buttons, initially holding a simple predefined formatting string acting as a placeholder ('%d - %d - %d'). The first button of the application lets you display a more complex sample format string in the edit box (the code has a simple assignment to the edit text of the format string 'value %d, Align %4d, Fill %4.4d'). The second button lets you apply the format string to the predefined values, using the following code:

```
var
  StrFmt: string;
  N1, N2, N3: Integer;
begin
  StrFmt := Edit1.Text;
  N1 := 8;
  N2 := 16;
  N3 := 256;

  Show (Format ('Format string: %s', [StrFmt]));
  Show (Format ('Input data: [%d, %d, %d]', [N1, N2, N3]));
  Show (Format ('Output: %s', [Format (StrFmt, [N1, N2, N3])]));
```


200 - 06: All About Strings

```
Show (' '); // blank line  
end;
```

If you display the output first with the initial format string and next with the sample format string (that is if you press the second button, the first, and then the second again), you should get an output like the following:

```
Format string: %d - %d - %d  
Input data: [8, 16, 256]  
Output: 8 - 16 - 256  
  
Format string: Value %d, Align %4d, Fill %4.4d  
Input data: [8, 16, 256]  
Output: Value 8, Align 16, Fill 0256
```

However the idea behind the program is to edit the format string and experiment with it, to see all of the various available formatting options.

The Internal Structure of Strings

While you can generally use strings without knowing much about their internals, it is interesting to have a look to the actual data structure behind this data type. In the early days of the Pascal language, strings had a maximum of 255 elements of one byte each and would use the first byte (or zero byte) for storing the string length. A lot of time has passed since those early days, but the concept of having some extra information about the string stored as part of its data remains a specific approach of the Object Pascal language (unlike many languages that derive from C and use the concept of a string terminator).

note `ShortString` is the name of the traditional Pascal string type, a string of one byte characters or `AnsiChar` limited to 255 characters. The `ShortString` type is still available in the desktop compilers, but not in the mobile ones. You can represent a similar data structure with a dynamic array of bytes, or `TBytes`, or a plain static arrays of `Byte` elements.

As I already mentioned, a string variable is nothing but a pointer to a data structure allocated on the heap. Actually, the value stored in the string is not a reference to the beginning of the data structure, but a reference to the first of the characters of the string, with string meta data data available at negative offsets from that location. The in-memory representation of the data of the string type is the following:

-12	-10	-8	-4	String reference address
Code page	Elem size	Ref count	Length	First char of string

The first element (counting backwards from the beginning of the string itself) is an Integer with the string length, the second element holds the reference count. Further fields (used on desktop compilers) are the element size in bytes (either 1 or 2 bytes) and the code page for older Ansi-based string types.

Quite surprisingly, it is possible to access most of these fields with specific low-level string meta data functions, beside the rather obvious `Length` function:

```
function StringElementSize(const S: string): word;
function StringCodePage(const S: string): word;
function StringRefCount(const S: string): Longint;
```

As an example, you can create a string and ask for some information about it, as I did in the `StringMetaTest` application project:

```
var
  Str1: string;
begin
  Str1 := 'F' + string.Create ('o', 2);

  Show ('SizeOf: ' + SizeOf (Str1).ToString);
  Show ('Length: ' + Str1.Length.ToString);
  Show ('StringElementSize: ' +
    StringElementSize (Str1).ToString);
  Show ('StringRefCount: ' +
    StringRefCount (Str1).ToString);
  Show ('StringCodePage: ' +
    StringCodePage (Str1).ToString);
  if StringCodePage (Str1) = DefaultUnicodeCodePage then
    Show ('Is Unicode');
  Show ('Size in bytes: ' +
    (Length (Str1) * StringElementSize (Str1)).ToString);
  Show ('ByteLength: ' +
    ByteLength (Str1).ToString);
```

note There is a specific reason the program builds the `'Foo'` string dynamically rather than assigning a constant, and that is because constant strings have the reference count disabled (or set to -1). In the demo I preferred showing a proper value for the reference count, hence the dynamic string construction.

This program produces output similar to the following:

```
SizeOf: 4
Length: 3
StringElementSize: 2
StringRefCount: 1
StringCodePage: 1200
Is Unicode
Size in bytes: 6
ByteLength: 6
```

202 - 06: All About Strings

The code page returned by a `UnicodeString` is 1200, a number stored in the global variable `DefaultUnicodeCodePage`. In the code above (and its output) you can clearly notice the difference between the size of a string variable (invariably 4), the logical length, and the physical length in bytes.

This can be obtained by multiplying the size in bytes of each character times the number of characters, or by calling `ByteLength`. This latter function, however, doesn't support some of the string types of the older desktop compiler.

Looking at Strings in Memory

The ability to look into a string's meta data can be used to better understand how string memory management works, particularly in relationship with the reference counting. For this purpose, I've added some more code to the `StringMetaTest` application project.

The program has two global strings: `MyStr1` and `MyStr2`. The program assigns a dynamic string to the first of the two variables (for the reason explained earlier in the note) and then assigns the second variable to the first:

```
MyStr1 := string.Create(['H', 'e', 'l', 'l', 'o']);  
MyStr2 := MyStr1;
```

Besides working on the strings, the program shows their internal status, using the following `StringStatus` function:

```
function StringStatus (const Str: string): string;  
begin  
    Result := 'Addr: ' + IntToStr (Integer (Str)) +  
             ', Len: ' + IntToStr (Length (Str)) +  
             ', Ref: ' + IntToStr (PInteger (Integer (Str) - 8)^) +  
             ', Val: ' + Str;  
end;
```

It is important in the `StringStatus` function to pass the string parameter as a `const` parameter. Passing this parameter by copy will cause the side effect of having one extra reference to the string while the function is being executed. By contrast, passing the parameter via a reference (`var`) or constant (`const`) doesn't imply a further reference to the string. In this case I've used a `const` parameter, as the function is not supposed to modify the string.

To obtain the memory address of the string (useful to determine its actual identity and to see when two different strings refer to the same memory area), I've simply made a hard-coded typecast from the string type to the `Integer` type. Strings are references-in practice, they're pointers: Their value holds the actual memory location of the string not the string itself.

The code used for testing what happens to the string is the following:

```
Show ('MyStr1 - ' + StringStatus (MyStr1));
Show ('MyStr2 - ' + StringStatus (MyStr2));
MyStr1 [1] := 'a';
Show ('Change 2nd char');
Show ('MyStr1 - ' + StringStatus (MyStr1));
Show ('MyStr2 - ' + StringStatus (MyStr2));
```

Initially, you should get two strings with the same content, the same memory location, and a reference count of 2.

```
MyStr1 - Addr: 51837036, Len: 5, Ref: 2, Val: Hello
MyStr2 - Addr: 51837036, Len: 5, Ref: 2, Val: Hello
```

As the application changes the value of one of the two strings (it doesn't matter which one), the memory location of the updated string will change. This is the effect of the copy-on-write technique. This is the second part of the output:

```
Change 2nd char
MyStr1 - Addr: 51848300, Len: 5, Ref: 1, Val: Hallo
MyStr2 - Addr: 51837036, Len: 5, Ref: 1, Val: Hello
```

You can freely extend this example and use the `StringStatus` function to explore the behavior of long strings in many other circumstances, with multiple references, when they are passed as parameters, assigned to local variables, and more.

Strings and Encoding

As we have seen the string type in Object Pascal is mapped to the Unicode UTF-16 format, with 2-bytes per element and management of surrogate pairs for code points outside of the BMP (Basic Multi-language Plane).

There are many cases, though, in which you need to save to file, load from file, transmit over a socket connection, or receive textual data from a connection that uses a different representation, like ANSI or UTF-8.

To convert files and in memory data among different formats (or encodings), the Object Pascal RTL has a handy `TEncoding` class, defined in the `System.Sysutils` unit along with several inherited classes.

note There are several other handy classes in the Object Pascal RTL that you can use for reading and writing data in text formats. For example, the `TStreamReader` and `TStreamWriter` classes offer support for text files with any encoding. These classes will be introduced in Chapter 18.

204 - 06: All About Strings

Although I still haven't introduced classes and inheritance, this set of encoding classes is very easy to use, as there is already a global object for each encoding, automatically created for you.

In other words, an object of each of these encoding classes is available within the `TEncoding` class, as a class property:

```
type
  TEncoding = class
  public
    class property ASCII: TEncoding read GetASCII;
    class property BigEndianUnicode: TEncoding
      read GetBigEndianUnicode;
    class property Default: TEncoding read GetDefault;
    class property Unicode: TEncoding read GetUnicode;
    class property UTF7: TEncoding read GetUTF7;
    class property UTF8: TEncoding read GetUTF8;
```

note The `Unicode` encoding is based on the `TUnicodeEncoding` class that uses the same UTF-16 LE (Little Endian) format used by the `string` type. The `BigEndianUnicode`, instead, uses the less common Big Endian representation. If you are not familiar with “Endianness” this is a terms used to indicate the sequence of two bytes making a code point (or any other data structure). Little Endian has the most significant byte first, and Big Endian has the most significant byte last. For more information, see en.wikipedia.org/wiki/Endianness.

Again, rather than exploring these classes in general, something a little difficult at this point of the book, let's focus on a couple of practical examples. The `TEncoding` class has methods for reading and writing Unicode strings to byte arrays, performing appropriate conversions.

To demonstrate UTF format conversions via `TEncoding` classes, but also to keep my example simple and focused and avoid working with the file system, in the `EncodingTest` application project I've created an UTF-8 string in memory using some specific data, and converted it to UTF-16 with a single function call:

```
var
  Utf8string: TBytes;
  Utf16string: string;
begin
  // process Utf8data
  SetLength (Utf8string, 3);
  Utf8string[0] := Ord ('a'); // single byte ANSI char < 128
  Utf8string[1] := $c9; // double byte, reversed latin a
  Utf8string[2] := $90;
  Utf16string := TEncoding.UTF8.GetString(Utf8string);
  Show ('Unicode: ' + Utf16string);
```

The output should be:

```
| Unicode: ae
```

Now to better understand the conversion and the difference in the representations, I've added the following code:

```
Show ('Utf8 bytes:');
for AByte in Utf8String do
  Show (AByte.ToString);

Show ('Utf16 bytes:');
UniBytes := TEncoding.Unicode.GetBytes (Utf16string);
for AByte in UniBytes do
  Show (AByte.ToString);
```

This code produces a memory dump, with decimal values, for the two representations of the string, UTF-8 (a one byte and a two byte code point) and UTF-16 (with both code points being 2 bytes):

```
Utf8 bytes:
97
201
144
Utf16 bytes:
97
0
80
2
```

Notice that direct character to byte conversion, for UTF-8, work only for ANSI-7 characters, that is values up to 127. For higher level ANSI characters there is no direct mapping and you must perform a conversion, using the specific encoding (which will however fail on multi-byte UTF-8 elements). So both of the following produce wrong output:

```
// error: cannot use char > 128
Utf8string[0] := Ord ('à');
Utf16string := TEncoding.UTF8.GetString(Utf8string);
Show ('wrong high ANSI: ' + Utf16string);
// try different conversion
Utf16string := TEncoding.ANSI.GetString(Utf8string);
Show ('wrong double byte: ' + Utf16string);

// output
wrong high ANSI:
wrong double byte: àÉ
```

The encoding classes let you convert in both directions, so in this case I'm converting from UTF-16 to UTF-8, doing some processing of the UTF-8 string (something to be done with care, given the variable length nature of this format), and convert back to UTF-16:

```
var
  Utf8string: TBytes;
  Utf16string: string;
  I: Integer;
```

206 - 06: All About Strings

```
begin
  Utf16string := 'This is my nice string with à and Æ';
  Show ('Initial: ' + Utf16string);

  Utf8string := TEncoding.UTF8.GetBytes(Utf16string);
  for I := 0 to High(Utf8string) do
    if Utf8string[I] = Ord('i') then
      Utf8string[I] := Ord('I');
  Utf16string := TEncoding.UTF8.GetString(Utf8string);
  Show ('Final: ' + Utf16string);
```

The output is:

```
Initial: This is my nice string with à and Æ
Final: This Is my nIce string with à and Æ
```

Other Types for Strings

While the string data type is by far the most common and largely used type for representing strings, Object Pascal desktop compilers had and still have a variety of string types. Some of these types can be used also on mobile applications, where you can also just use `TBytes` directly to manipulate string with a 1-byte representation, as in the application project described in the last section.

While developers who used Object Pascal in the past might have a lot of code based on these pre-Unicode types (or directly managing UTF-8), modern applications really require full Unicode support. Also while some types, like `UTF8String`, are available in the language, their support in terms of RTL is limited. The recommendation is to use plain and standard Unicode strings.

note There has been a lot of discussion and criticism about the original lack of native types like `AnsiString` and `UTF8String` in the Object Pascal mobile compilers. In Delphi 10.1 Berlin the `UTF8String` type and the low-level `RawByteString` type have been officially re-introduced and later Delphi 10.4 enabled all desktop string types also on mobile. It is still worth considering that there is almost no other programming language out there that has more than one native or intrinsic string type. Multiple string types are more complex to master, can cause unwanted side effects (like extensive automatic conversion calls that slow down programs), and cost a lot for the maintenance of multiple versions of all of the string management and processing functions. So the recommendation, beside corner cases, is to focus on the standard string type, or `UnicodeString`.

The UCS4String type

An interesting but little used string type is the UCS4String type, available on all compilers. This is just an UTF32 representation of a string, and no more than an array of UTF32Char elements, or 4-bytes characters. The reason behind this type, as mentioned earlier, is that it offers a direct representation of all of the Unicode code points. The obvious drawback is such a string takes twice as much memory than a UTF-16 string (which already takes twice as much than an ANSI string).

Although this data type can be used in specific situations, it is not particularly suited for general circumstances. Also, this type doesn't support copy-on-write nor has any real system functions and procedures for processing it.

note Whilst the UCS4String guarantees one UTF32Char per Unicode code point, it cannot guarantee one UTF32Char per grapheme, or “visual character”.

Older String Types

As mentioned, the Object Pascal compiler offers support for some older, traditional string types (and these are available on all target platforms starting with Delphi 10.4). These older string types include:

- The `shortString` type, which corresponds to the original Pascal language string type. These strings have a limit of 255 characters. Each element of a short string is of type `ANSIChar`.
- The `ANSIString` type, which corresponds to variable-length strings. These strings are allocated dynamically, reference counted, and use a copy-on-write technique. The size of these strings is *almost* unlimited (they can store up to two billion characters!). Also this string type is based on the `ANSIChar` type. Available also on mobile compilers, even if the ANSI representation is specific to Windows and some special characters might be handled differently depending on the platform.
- The `wideString` type is similar to a 2-bytes Unicode string in terms of representation, is based on the `char` type, but unlike the standard string type it doesn't use copy-on-write and it is less efficient in terms of memory allocation. If you wonder why it was added to the language, the reason was for compatibility with string management in Microsoft's COM architecture.
- `UTF8String` is a string based on the variable character length UTF-8 format. As I mentioned there is little run-time library support for this type.

208 - 06: All About Strings

- `RawByteString` is an array of characters with no code page set, on which no character conversion is ever accomplished by the system (thus logically resembling a `TBytes` structure, but allowing some direct string operations that an array of bytes currently lacks). This data type should be rarely used outside of libraries.
- A string construction mechanism allowing you to define a 1-byte string associated with a specific ISO code page, a remnant of the pre-Unicode past.

Again, all of these string types can be used on desktop compilers, but are available only for backwards compatibility reason. The goal is to use Unicode, `TEncoding`, and other modern string management techniques whenever possible.

part ii: oop in object pascal

Many modern programming languages support some form of *object-oriented programming* (OOP) paradigm. Many of them use a class-based one that is based on three fundamental concepts:

- Classes, data types with a public interface and a private data structure, implementing encapsulation; instances of these data types are generally called objects,
- Class extensibility or inheritance, which is the ability to extend a data type with new features without modifying the original one,
- Polymorphism or late binding, which is the ability to refer to objects of different classes with a uniform interface, and still operate on objects in the way defined by their specific type.

note Other languages such as IO, JavaScript, Lua and Rebol use a prototype based object-oriented paradigm, where objects can be created from other objects rather than from a class depending on how the object is created. They do provide a form of inheritance, but from another object rather than a class, and dynamic typing that can be used to implement polymorphism, even if in a rather different way.

210 - Part II: OOP in Object Pascal

You can write Object Pascal applications even without knowing a lot about object oriented programming. As you create a new form, add new components, and handle events, the IDE prepares most of the related code for you automatically. But knowing the details of the language and its implementation will help you understand precisely what the system is doing and allow you to master the language completely.

You will also be able to create complex architectures within your applications, and even entire libraries, and embrace and extend the components that come with the development environment.

The second part of the book is focused on core object-oriented programming (OOP) techniques. The aim of this part of the book is both to teach the fundamental concepts of OOP and to detail how Object Pascal implements them, comparing it with other similar OOP languages.

Summary of Part II

Chapter 7: Objects

Chapter 8: Inheritance

Chapter 9: Handling Exceptions

Chapter 10: Properties and Events

Chapter 11: Interfaces

Chapter 12: Manipulating Classes

Chapter 13: Objects and Memory

07: objects

Even if you don't have a detailed knowledge of object-oriented programming (OOP), this chapter will introduce each of the key concepts. If you are already fluent in OOP, you can probably go through the material relatively quickly and focus on Object Pascal language specifics, in comparison to other languages you might already know.

The OOP support in Object Pascal has a lot of similarities to languages like C# and Java, it also has some resemblances with C++ and other static and strongly-typed languages. Dynamic languages, instead, tend to offer a different interpretation of OOP, as they treat the type system in a more loose and flexible way.

note A lot of the conceptual similarities between C# and Object Pascal are due to the fact that the two languages share the same designer, Anders Hejlsberg. He was the original author of the Turbo Pascal compilers, of the first version of Delphi's Object Pascal, and later moved to Microsoft and designed C# (and more recently the JavaScript derivative TypeScript). You can read more about the Object Pascal language history in Appendix A.

Introducing Classes and Objects

Class and *object* are two terms commonly used in Object Pascal and other OOP languages. However, because they are often misused, let's be sure we agree on their definitions from the very beginning:

- A *class* is a user-defined data type, which defines a state (or a representation) and some operations (or behaviors). In other terms, a class has some internal data and some methods, in the form of procedures or functions. A class usually describes the characteristics and behavior of a number of similar objects, although there are special purpose classes that are meant for a single object.
- An *object* is an instance of a class, that is a variable of the data type defined by the class. Objects are *actual* entities. When the program runs, objects take up some memory for their internal representation.

The relationship between an object and a class is the same as the one between any other variable and its data type. Only, in this case variables have a special name.

history The OOP terminology dates back to the first few languages that adopted the model, like Smalltalk. Most of the original terminology, however, was later dropped in favor of terms in use in procedural languages. So while terms like classes and objects are still commonly used, you'd generally hear the term invoking a method more often than the original term sending a message to a receiver (an object). A full and detailed guide to the OOP jargon and how it evolved over time could be interesting, but would take too much space in this book.

The Definition of a Class

In Object Pascal you can use the following syntax to define a new class data type (TDate), with some local data fields (Month, Day, Year) and some methods (SetValue, LeapYear):

```
type
  TDate = class
    FMonth, FDay, FYear: Integer;
    procedure SetValue (M, D, Y: Integer);
    function LeapYear: Boolean;
  end;
```

note We have already seen a similar structure for records, which are quite similar to classes in term of definition. There are differences in memory management and other areas, as detailed later in this chapter. Historically, though, in Object Pascal this syntax was first adopted for classes and later ported back to records.

The convention in Object Pascal is to use the letter *T* as a prefix for the name of every class you write, like for any other type (*T* stands for *Type*, in fact). This is just a convention—to the compiler, *T* is just a letter like any other—but it is so common that following it will make your code easier to understand by other programmers.

Unlike other languages, the class definition in Object Pascal doesn't include the actual implementation (or definition) of the methods, but only their signature (or declaration). This makes the class code more compact and significantly more readable.

tip Although it might look like getting to the actual implementation of the method is more time consuming, the editor allows you to use the combination of the Shift+Up and Shift+Down keys to navigate from the method declarations to their implementations and vice versa. Moreover, you can let the editor generate a skeleton of the definition of the methods, after you write the class definition, by using Class Completion (pressing the Ctrl+C keys while the cursor is within the class definition).

Also keep in mind that beside writing the definition of a class (with its fields and methods) you can also write a declaration. This has only the class name, as in:

```
type
  TMyDate = class;
```

The reason for such a declaration lies in the fact that you might need to have two classes referencing each other. Given in Object Pascal you cannot use a symbol until it is defined, to refer a not-yet-defined class you need a declaration. I wrote the following code fragment only to show you the syntax, not that it makes any sense:

```
type
  THusband = class;

  TWife = class
    FHusband: THusband;
  end;

  THusband = class
    FWife: TWife;
  end;
```

You'll encounter similar cross-references in real code, which is why this syntax is important to keep in mind. Notice, that like for methods, a class declared in a unit must be fully defined later in the same unit.

Classes in Other OOP Languages

As a comparison, this is the `TDate` class written in C# and in Java (which in this simplified case happen to be the same) using a more appropriate set of naming rules, with the code of the methods omitted:

```
// C# and Java language

class Date
{
    int    month;
    int    day;
    int    year;

    void setValue (int m, int d, int y)
    {
        // code
    }

    bool leapYear()
    {
        // code
    }
}
```

In Java and C# the methods' code comes within the class definition, while in Object Pascal the methods declared in a class should be fully defined in the implementation portion of the same unit that includes the class definition. In other words, in Object Pascal a class is always completely defined in a single unit (while a unit can, of course, contain multiple classes). By contrast, while in C++ methods are separately implemented like in Object Pascal, but a header file containing a class definition has no strict correspondence to an implementation file with the method's code. A corresponding C++ class would look like:

```
// C++ language

class Date
{
    int    month;
    int    day;
    int    year;

    void setValue (int m, int d, int y);
    bool leapYear();
}
```

The Class Methods

Like with records, when you define the code of a method you need to indicate which class it is part of (in this example the `TDate` class) by using the class name as a prefix and the dot notation, as in the following code:

```
procedure TDate.SetValue(M, D, Y: Integer);
begin
    FMonth := M;
    FDay := D;
    FYear := Y;
end;

function TDate.LeapYear: Boolean;
begin
    // call IsLeapYear in SysUtils.pas
    Result := IsLeapYear (FYear);
end;
```

Differently from most other OOP languages that define methods as functions, Object Pascal brings over the core distinction between procedures and functions, depending on the presence of a return value, also for methods. This is not the case in C++, where a separately defined method implementation would look like:

```
// C++ method
void Date::setValue(int m, int d, int y)
{
    month = m;
    day = d;
    year = y;
};
```

Creating an Object

After this comparison with other popular languages, let's get back to Object Pascal to see how you can use a class. Once the class has been defined, we can create an object of this type and use it as in the following code snippet (extracted from the `Dates1` application project like all of the code in this section):

```
var
    ADay: TDate;
begin
    // create
    ADay := TDate.Create;
    // use
    ADay.SetValue (1, 1, 2020);
    if ADay.LeapYear then
        Show ('Leap year: ' + ADay.Year.ToString);
```


The notation used is nothing unusual, but it is powerful. We can write a complex function (such as `LeapYear`) and then access its value for every `TDate` object as if it were a primitive data type. Notice that `ADay.LeanYear` is an expression similar to `ADay.Year`, although the first is a function call and the second a direct data access. As we'll see in Chapter 10, the notation used by Object Pascal to access properties is again the same.

note Calls of methods with no parameters in most programming languages based on the C language syntax require parenthesis, like in `ADay.LeanYear()`. This syntax is legal also in Object Pascal, but rarely used. Methods with no parameters are generally called without the parenthesis. This is very different from many languages in which a reference to a function or method with no parenthesis returns the function address. As we have seen in the section “Procedural Types” in Chapter 4, Object Pascal uses the same notation for calling a function or reading its address, depending on the context of the expression.

The output of the code snippet above is fairly trivial:

```
| Leap year: 2020
```

Again, let me compare the object creation with similar code written in other programming languages:

```
| // C# and Java languages (object reference model)
| Date aDay = new Date();
|
| // C++ language (two alternative styles)
| Date aDay; // local allocation
| Date* aDay = new Date(); // "manual" reference
```

The Object Reference Model

In some OOP languages like C++, declaring a variable of a class type creates an instance of that class (more or less like it happens with records in Object Pascal). The memory for a local object is taken from the stack, and released when the function terminates. In most cases, though, you have to explicitly use pointers and references to have more flexibility in managing the lifetime of an object, adding a lot of extra complexity.

The Object Pascal language, instead, is based on an *object reference model*, exactly like Java or C#. The idea is that each variable of a class type does not hold the actual value of the object with its data (to store the day, month, and year, for example). Rather, it contains only a reference, or a *pointer*, to indicate the memory location where the actual object data is stored.

note In my opinion, adopting the object reference model was one of the best design decisions made by the compiler team in the early days of the language, when this model wasn't so common in programming languages (in fact, at the time Java wasn't available and C# didn't exist).

This is why in these languages you need to explicitly create an object and assign it to a variable, as objects are not automatically initialized. In other words, when you declare a variable, you don't create an object in memory, you only reserve the memory location for a reference to the object. Object instances must be created manually and explicitly, at least for the objects of the classes you define. (In Object Pascal, though, instances of components you place on a form are built automatically by the run time library.)

In Object Pascal, to create an instance of an object, we can call its special `Create` method, which is a constructor or another custom constructor defined by the class itself. Here is the code again:

```
■ ADay := TDate.Create;
```

As you can see, the constructor is applied to the class (the type), not to the object (the variable). That's because you are asking the class to create a new instance of its type, and the result is a new object you'd generally assign to a variable.

Where does the `Create` method come from? It is a constructor of the class `TObject`, from which all the other classes inherit (a topic covered in the next chapter). It is very common to add custom constructors to your classes, though, as we'll see later in this chapter.

Disposing Objects

In languages that use an object reference model, you need a way to create an object before using it, and you also need a means of releasing the memory it occupies when it is no longer needed. If you don't dispose of it, you end filling up memory with objects you don't need any more, causing a problem known as a *memory leak*. To solve this issue languages like C# and Java, based on a virtual execution environment (or virtual machine) adopt garbage collection. While this make developer's life easier, however this approach is subject to some complex performance-related issues that it isn't really relevant in explaining Object Pascal. So, interesting as the issues are, I don't want to delve into them here.

In Object Pascal, you generally release the memory of an object by calling its special `Free` method (again, a method of `TObject`, available in each and every class). `Free` removes the object from memory after calling its destructor (which can have special clean up code). So you can complete the code snippet above as:

218 - 07: Objects

```
var
  ADay: TDate;
begin
  // create
  ADay := Tdate.Create;

  // use (code missing)

  // free the memory
  ADay.Free;
end;
```

While this is the standard approach, the component library adds concepts like object ownership to significantly lessen the impact of manual memory management, making this a relatively simple issue to handle.

note As we'll see later, when using interfaces referring to objects the compiler adopts a form of Automatic Reference Counting (ARC) memory management. For a few years, this was also used for regular class type variables in Delphi mobile compilers. Starting with version 10.4 Sydney, the memory management model was unified adopting the classic, desktop Delphi memory management for all target platforms.

There is much more to memory management that you need to know, but given this is a rather important topic and not a simple one, I decided to offer only a short introduction here and have a full chapter focused on this topic, namely Chapter 13. In that chapter, I'll show you in details different techniques you can use.

What is “Nil”?

As I've mentioned, a variable can refer to an object of a given class. But it might not be initialized yet, or the object it used to refer to might not be available any longer. This is where you can use `nil`. This is a constant value indicating that the variable is not assigned to any object (or it is assigned to a 0 memory location). Other programming language use the symbol `null` to express the same concept.

When a variable of a class type has no value, you can initialize it this way:

```
| ADay := nil;
```

To check if an object has been assigned the variable, you can write either of the following expressions:

```
| if ADay <> nil then ...
| if Assigned (ADay) then ...
```

Do not make the mistake of assigning `nil` to an object to remove it from memory. Setting an object reference to `nil` and freeing it are two different operations. So you

often need to both free an object and set its reference to `nil`, or call a special purpose procedure that does both operations at once, called `FreeAndNil`. Again, more information and some actual demos will be coming in Chapter 13, focused on memory management.

Records vs. Classes in Memory

As I've mentioned earlier, one of the main differences between records and objects relates to their memory model. Record type variables use local memory, they are passed as parameters to functions by value by default, and they have a “copy by value” behavior on assignments. This contrasts with class type variables that are allocated on the dynamic memory heap, are passed by reference, and have a “copy by reference” behavior on assignments (thus copying the reference to the same object in memory, not the actual data).

note A consequence of this different memory management is that records lack inheritance and polymorphisms, two features we'll be focusing on in the next chapter.

- The `private` access specifier denotes fields and methods of a class that are not accessible outside the unit (the source code file) that declares the class.

For example, when you declare a record variable on the stack, you can start using it right away, without having to call its constructor (unless they are custom managed records). This means record variables are leaner and more efficient on the memory manager than regular objects, as they do not participate in the management of the dynamic memory. These are the key reasons for using records instead of objects for small and simple data structures.

Regarding the difference in the way records and objects are passed as parameters, consider that the default is to make a full copy of the memory block representing the record (including all of its data) or of the reference to the object (while the data is not copied). Of course, you can use `var` or `const` record parameters to modify the default behavior for passing record type parameters avoiding any copy.

Private, Protected, and Public

- The `strict private` access specifier denotes fields and methods that are not accessible outside of any method of the class, including methods of other classes

in the same unit. This matches the behavior of the `private` keyword in most other OOP languages.

A class can have any amount of data fields and any number of methods. However, for a good object-oriented approach, data should be hidden, or *encapsulated*, inside the class using it. When you access a date, for example, it makes no sense to change the value of the day by itself. In fact, changing the value of the day might result in an invalid date, such as February 30th. Using methods to access the internal representation of an object limits the risk of generating erroneous situations, as the methods can check whether the date is valid and refuse to modify the new value if it is not. Proper encapsulation is particularly important because it gives the class writer the freedom to modify the internal representation in a future version.

The concept of encapsulation is quite simple: just think of a class as a “black box” with a small, visible portion. The visible portion, called the *class interface*, allows other parts of a program to access and use the objects of that class. However, when you use the objects, most of their code is hidden. You seldom know what internal data the object has, and you usually have no way to access the data directly. Rather you use the methods to access the data of an object or act on it.

Encapsulation using `private` and `protected` members is the object-oriented solution to a classic programming goal known as *information hiding*.

Object Pascal has five basic access (or visibility) specifiers: `private`, `protected`, and `public`. A sixth one, `published`, will be discussed in the Chapter 10. Here are the five basic ones:

- The `public` access specifier denotes fields and methods that are freely accessible from any other portion of a program as well as in the unit in which they are defined.
- The `protected` and `strict protected` access specifier are used to indicate methods and fields with limited visibility. Only the current class and its derived classes (or subclasses) can access `protected` elements unless they are in the same class or in any case depending on the `strict` modifier. We’ll discuss this keyword again in the “Protected Fields and Encapsulation” section of the next chapter.

Generally, the fields of a class should be `private` or `strict private`; the methods are usually `public`. However, this is not always the case. Methods can be `private` or `protected` if they are needed only internally to perform some partial operations. Fields can be `protected` if you are fairly sure that their type definition is not going to change and you might want to manipulate them directly in derived classes (as explained in the next chapter), although this is rarely recommended.

As a general rule, you should invariably avoid `public` fields, and generally expose some direct way to access the data using properties, as we'll see in detail in Chapter 10. Properties are an extension to the encapsulation mechanism of other OOP languages and are very important in Object Pascal.

As mentioned, the `private` access specifiers only restrict code outside a unit from accessing certain members of classes declared in that unit. This means that if two classes are in the same unit, there is no protection for their private fields unless they are marked `strict private`, which is generally a good idea.

note The C++ language has the concept of friend classes, that is classes allowed to access another class private data. Following this terminology, we can say that in Object Pascal all classes in the same unit are automatically considered as friend classes.

An Example of Private Data

As an example of the use of these access specifiers for implementing encapsulation, consider this new version of the `TDate` class:

```
type
  TDate = class
    private
      Month, Day, Year: Integer;
    public
      procedure SetValue (M, D, Y: Integer);
      function LeapYear: Boolean;
      function GetText: string;
      procedure Increase;
    end;
```

In this version, the fields are now declared to be `private`, and there are some new methods. The first, `GetText`, is a function that returns a string with the date. You might think of adding other functions, such as `GetDay`, `GetMonth`, and `GetYear`, which simply return the corresponding `private` data, but similar direct data-access functions are not always needed. Providing access functions for each and every field might reduce the encapsulation, weaken the abstraction, and make it harder to modify the internal implementation of a class later on. Access functions should be provided only if they are part of the logical interface of the class you are implementing, not because there are matching fields.

The second new method is the `Increase` procedure, which increases the date by one day. This is far from simple, because you need to consider the different lengths of the various months as well as leap and non-leap years. What I'll do to make it easier to write the code is to change the internal implementation of the class to use Object

222 - 07: Objects

Pascal `TDateTime` type for the internal implementation. So the actual class will change to the following code you can find in the `Dates2` application project:

```
type
  TDate = class
  private
    FDate: TDateTime;
  public
    procedure SetValue (M, D, Y: Integer);
    function LeapYear: Boolean;
    function GetText: string;
    procedure Increase;
  end;
```

Notice that because the only change is in the `private` portion of the class, you won't have to modify any of your existing programs that use it. This is the advantage of encapsulation!

note In this new version of the class, the (only) field has an identifier that starts with the letter "F". This is a fairly common convention in Object Pascal and one I'll generally use in the book.

To end this section, let me finish describing the project, by listing the source code of the class methods, which rely on a few system functions for mapping dates to the internal structure and vice verse:

```
procedure TDate.SetValue (M, D, Y: Integer);
begin
  FDate := EncodeDate (Y, M, D);
end;

function TDate.GetText: string;
begin
  Result := DateToStr (FDate);
end;

procedure TDate.Increase;
begin
  FDate := FDate + 1;
end;

function TDate.LeapYear: Boolean;
begin
  // call IsLeapYear in Sysutils and YearOf in DateUtils
  Result := IsLeapYear (YearOf (FDate));
end;
```

Notice also how the code to use the class cannot refer to the `Year` value any more, but it can only return information about the date object as allowed by its methods:

```
var
  ADay: TDate;
begin
```

```

// create
ADay := TDate.Create;

// use
ADay.SetValue (1, 1, 2020);
ADay.Increase;

if ADay.LeapYear then
  Show ('Leap year: ' + ADay.GetText);

// free the memory (for non ARC platforms)
ADay.Free;

```

The output is not much different than before:

```

| Leap year: 1/2/2020

```

Encapsulation and Forms

One of the key ideas of encapsulation is to reduce the number of global variables used by a program. A global variable can be accessed from every portion of a program. For this reason, a change in a global variable affects the whole program. On the other hand, when you change the representation of a field of a class, you only need to change the code of some methods of that class referring to the given field, and nothing else. Therefore, we can say that information hiding refers to *encapsulating changes*.

Let me clarify this idea with a practical example. When you have a program with multiple forms, you can make some data available to every form by declaring it as a global variable in the interface portion of the unit of the form:

```

var
  Form1: TForm1;
  NClicks: Integer;

```

This works but has two problems. First, the data (`NClicks`) is not connected to a specific instance of the form, but to the entire program. If you create two forms of the same type, they'll share the data. If you want every form of the same type to have its own copy of the data, the only solution is to add it to the form class:

```

type
  TForm1 = class(TForm)
  public
    FNClicks: Integer;
  end;

```

The second problem is that if you define the data as a global variable or as a public field of a form, you won't be able to modify its implementation in the future without affecting the code that uses the data. For example, if you only have to read the cur-

224 - 07: Objects

rent value from other forms, you can declare the data as private and provide a method to read the value:

```
type
  TForm1 = class(TForm)
    // components and event handlers here
  public
    function GetClicks: Integer;
  private
    FNClicks: Integer;
  end;

function TForm1.GetClicks: Integer;
begin
  Result := FNClicks;
end;
```

An even better solution is to add a property to the form, as we'll see in Chapter 10. You can experiment with this code by opening the ClicksCount application project. In short, the form of this project has two buttons and a label at the top, with most of the surface empty for a user to click (or tap) onto it. In this case, the count is increased and the label is updated with the new value:

```
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;
  Shift: TShiftState; X, Y: Single);
begin
  Inc (FNClicks);
  Label1.Text := FNClicks.ToString;
end;
```

You can see the application in action in Figure 7.1. The project's form also has two buttons, one for creating a new form of the same type and the second to close it (so you can give focus back to the previous form).

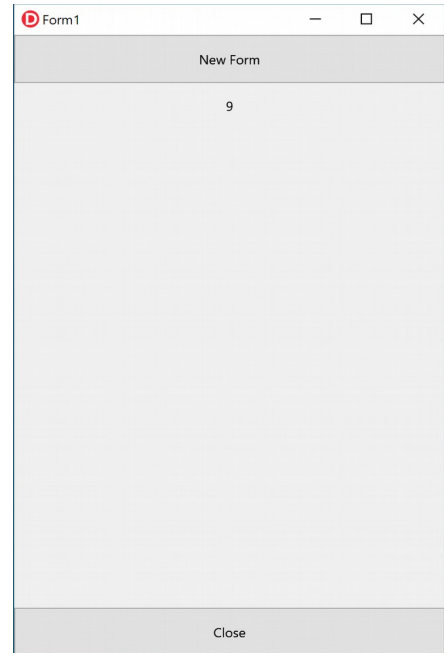
This is done to emphasize how different instances of the same form type each have their own clicks count. This is the code of the two methods:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  NewForm: TForm1;
begin
  NewForm := TForm1.Create(Application);
  NewForm.Show;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  Close;
end;
```

Figure 7.1:

The form of the ClicksCount application project showing the number of clicks or taps on a form (tracked using private form data)



The Self Keyword

We've seen that methods are very similar to procedures and functions. The real difference is that methods have an extra, implicit parameter. This is a reference to the current object, the object the method is applied to. Within a method you can refer to this parameter—the current object—using the `self` keyword.

This extra hidden parameter is needed when you create several objects of the same class, so that each time you apply a method to one of the objects, the method will operate only on the data of that specific objects and not affect the other objects of the same class.

note We have already seen the role of the `self` keywords in Chapter 5, while discussing records. The concept and its implementation are very similar. Again, historically `self` was first introduced for classes and later extended to records, when methods were added also to this data structure.

226 - 07: Objects

For example, in the `SetValue` method of the `TDate` class, listed earlier, we simply use `Month`, `Year`, and `Day` to refer to the fields of the current object, something you might express as:

```
Self.FMonth := M;  
Self.FDay := D;
```

This is actually how the Object Pascal compiler translates the code, *not* how you are supposed to write it. The `Self` keyword is a fundamental language construct used by the compiler, but at times it is used by programmers to resolve name conflicts and to make code more readable.

note The C++, Java, C#, and JavaScript languages have a similar feature based on the keyword `this`. However in JavaScript using `this` in a method to refer to object fields is compulsory, unlike in C++, C# and Java.

All you really need to know about `Self` is that the technical implementation of a call to a method differs from that of a call to a generic subroutine. Methods have that extra hidden parameter, `Self`. Because all this happens behind the scenes, you don't need to know how `Self` works at this time.

The second important thing to know is that you can explicitly use `Self` to refer to the current object as a whole, for example passing the current object as parameter to another function.

Creating Components Dynamically

As an example of what I've just mentioned, the `Self` keyword is often used when you need to refer to the current form explicitly in one of its methods.

A typical example is the creation of a component at run time, where you must pass the owner of the component to its `Create` constructor and assign the same value to its `Parent` property. In both cases, you have to supply the current form object as parameter or value, and the best way to do this is to use the `Self` keyword.

note The ownership of a component indicates a lifetime and memory management relationship between two objects. When the owner of a component is freed the component will also be freed. Parenthood refers to visual controls hosting a child control within their surface.

To demonstrate this kind of code, I've written the `CreateComps` application project. This application has a simple form with no components and a handler for its `OnMouseDown` event, which also receives as its parameter the position of the mouse click. I need this information to create a button component in that position.

note An event handler is a special method covered in Chapter 10, and part of the same family of the button's `OnClick` event handlers we have already used in this book.

Here is the code of the method:

```
procedure TForm1.FormMouseDown (Sender: TObject;
    Button: TMouseButton; Shift: TShiftState; X, Y: Integer);
var
    Btn: TButton;
begin
    Btn := TButton.Create (Self);
    Btn.Parent := Self;
    Btn.Position.X := X;
    Btn.Position.Y := Y;
    Btn.Height := 35;
    Btn.Width := 135;
    Btn.Text := Format ('At %d, %d', [X, Y]);
end;
```

Notice you might need to add the `StdCtrls` unit to the uses statement to compile this event handler.

The effect of this code is to create buttons at mouse-click positions, with a caption indicating the exact location, as you can see in Figure 7.2. (For this project, I disabled the FMX Mobile Preview to show native-styled Windows buttons, as it is more clear.) In the code above, notice in particular the use of the `Self` keyword, as the parameter of the `Create` method and as the value of the `Parent` property.

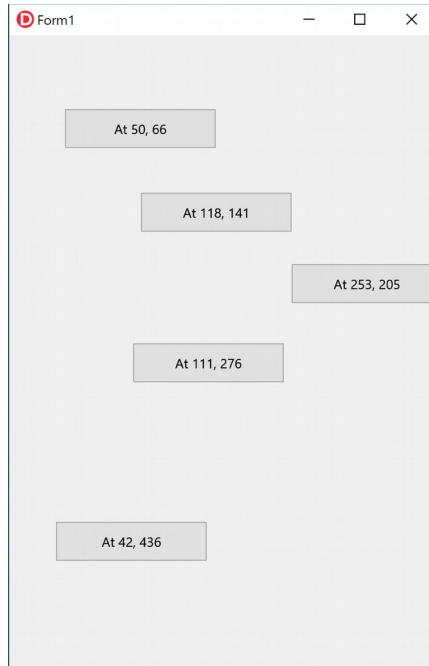
When writing a procedure like the code you've just seen, you might be tempted to use the `Form1` variable instead of `Self`. In this specific example, that change wouldn't make any practical difference (although it won't be good coding practice), but if there are multiple instances of a form, using `Form1` would really be an error.

In fact, if the `Form1` variable refers to a form of that type being created (generally the first one) and if you create two instances of the same form type, by clicking on any following forms the new button will always be displayed in the first one. Its owner and `Parent` will be `Form1` and not the form on which the user has clicked.

In general, writing a method in which you refer to a particular instance of the same class when the current object is required is a really a bad OOP coding style.

Figure 7.2:

The output of the CreateComps application project example on a mobile device



Constructors

In the code above, to create an object of a class (or allocate the memory for an object), I've called the `Create` method. This is a *constructor*, a special method that you can apply to a class to allocate memory for a new instance of that class:

```
| ADay := TDate.Create;
```

The instance is returned by the constructor and can be assigned to a variable for storing the object and using it later on. When you are creating an object, its memory is initialized. All of the data of the new instance to set to zero (or `nil`, or empty string, or the proper “default” value for a given data type).

If you want your instance data to start out with a nonzero value (particularly when a zero value makes little sense as a default), you need to write a custom constructor to do that. The new constructor can be called `Create`, or it can have any other name. What determines its role is not the name but the use of a constructor keyword.

note In other words, Object Pascal supports named constructors, while in many OOP languages the constructor must be named after the class itself. With named constructors, you can have more than one constructor with the same parameters (beside overloading the `Create` symbol – overloading is covered in the next section). Another very special feature of the language, quite unique among OOP languages, is that constructors can also be virtual. I'll show some examples that cover the consequences of this very nice feature later in the book, after introducing the concept of a virtual method in the next chapter.

The main reason to add a custom constructor to a class is to initialize its data. If you create objects without initializing them, calling methods later on may result in odd behavior or even a run-time error. Instead of waiting for these errors to appear, you should use preventive techniques to avoid them in the first place. One such technique is the consistent use of constructors to initialize objects' data. For example, we must call the `SetValue` procedure of the `TDate` class after we've created the object. As an alternative, we can provide a customized constructor, which creates the object and gives it an initial value:

```
constructor TDate.Create;
begin
    FDate := Today;
end;

constructor TDate.CreateFromValues (M, D, Y: Integer);
begin
    FDate := SetValue (M, D, Y);
end;
```

You can use these constructors as follows, as I've done in the `Date3` application project, in the code attached to two separate buttons:

```
Aday1 := TDate.Create;
Aday2 := TDate.CreateFromValues (12, 25, 2015);
```

Although in general you can use any name for a constructor, keep in mind that if you use a name other than `Create`, the `Create` constructor of the base `TObject` class will still be available. If you are developing and distributing code for others to use, a programmer calling this default `Create` constructor might bypass the initialization code you've provided. By defining a `Create` constructor with some parameters (or none, as in the example above), you replace the default definition with a new one and make its use compulsory.

In the same way that a class can have a custom constructor, it can have a custom destructor, a method declared with the `destructor` keyword and invariably called `Destroy`. This destructor method which can perform some resource cleanup before an object is destroyed, but in many cases a custom destructor is not required.

Just as a constructor call allocates memory for the object, a destructor call frees the memory. Custom destructors are really only needed for objects that acquire resources, such as another object, in their constructors or during their lifetime.

Differently from the default `Create` constructor, the default `Destroy` destructor is virtual and it is highly recommended that developer override this virtual destructor (virtual methods are covered in the next chapter).

That's because instead of a calling destructor directly to free an object, it is a good a common Object Pascal programming practice to call the special `Free` method of the `TObject` class, which in turn calls `Destroy` only if the object exists—that is, if it is not `nil`. So, if you define a destructor with a different name, it won't be called by `Free`. Again, more on this topic when we'll focus on memory management in Chapter 13.

note As covered in the next chapter, `Destroy` is a virtual method. You can replace its base definition with a new one in an inherited class marking it with the `override` keyword. By the way, having a static method that calls a virtual one is a very common programming style, called the *template pattern*. In a destructor, you should generally only write resource cleanup code. Try to avoid more complex operations, likely to raise exceptions or to take a significant amount of time, to avoid trouble in objects cleanup and because many destructors are called on program termination and you want to keep it fast.

Managing Local Class Data with Constructors and Destructors

Even if I'll cover more complex scenarios later in the book, here I want to show you a simple case of resource protection using a constructor and a destructor. This is the most common scenario for using a destructor. Suppose you have a class with the following structure (also part of the `Date3` application project):

```
type
  TPerson = class
  private
    FName: string;
    FBirthDate: TDate;
  public
    constructor Create (Name: string);
    destructor Destroy; override;
    // some actual methods
    function Info: string;
  end;
```

This class has a reference to another, internal object called `FBirthDate`. When an instance of the `TPerson` class is created, this internal (or child) object should also be created, and when the instance is destroyed, the child object should also be dis-

posed of. Here is how you can write the code of the constructor and overridden destructor, and of the internal method that can always take for granted that the internal object exists:

```
constructor TPerson.Create (Name: string);
begin
    FName := Name;
    FBirthDate := TDate.Create;
end;

destructor TPerson.Destroy;
begin
    FBirthDate.Free;
    inherited;
end;

function TPerson.Info: string;
begin
    Result := FName + ': ' + FBirthDate.GetText;
end;
```

note To understand the `override` keyword used to define the destructor and the `inherited` keyword within its definition, you'll have to wait until the next chapter. For now suffice to say the first is used to indicate that the class has a new definition replacing the base `Destroy` destructor, while the latter is used to invoke that base class destructor. Notice also that `override` is used in the method declaration, but not in the method implementation code.

Now you can use an object of the external class as in the following scenario, and the internal object will be properly created when the `TPerson` object is created and destroyed in a timely fashion when `TPerson` is destroyed:

```
var
    Person: TPerson;
begin
    Person := TPerson.Create ('John');
    // use the class and its internal object
    Show (Person.Info);
    Person.Free;
end;
```

Again, you can find this code as part of the `Dates3` application project.

Overloaded Methods and Constructors

Object Pascal supports overloaded functions and methods: you can have multiple methods with the same name, provided that the parameters are different. We have already seen how overloading works for global functions and procedures the same

232 - 07: Objects

rules apply to methods. By checking the parameters, the compiler can determine which version of the method you want to call.

Again, there are two basic rules for overloading:

- Each version of the method must be followed by the `overload` keyword.
- The differences must be in the number or type of the parameters or both. The return type, instead, cannot be used to distinguish among two methods.

If overloading can be applied to all of the methods of a class, this feature is particularly relevant for constructors, because we can have multiple constructors and call them all `Create`, which makes them easy to remember.

history Historically, overloading was added to C++ specifically to allow the use of multiple constructors, given they must have the same name (the name of the class). In Object Pascal, this feature could have been considered unnecessary, simply because multiple constructors can have different specific names, but it was added to the language anyway as it also useful in many other scenarios.

As an example of overloading, I've added to the `TDate` class two different versions of the `SetValue` method:

```
type
  TDate = class
  public
    procedure SetValue (Month, Day, Year: Integer); overload;
    procedure SetValue (NewDate: TDateTime); overload;

  procedure TDate.SetValue (Month, Day, Year: Integer);
  begin
    FDate := EncodeDate (Year, Month, Day);
  end;

  procedure TDate.SetValue(NewDate: TDateTime);
  begin
    FDate := NewDate;
  end;
```

After this simple step, I've added to the class two separate `Create` constructors, one with no parameters, which hides the default constructor, and one with the initialization values. The constructor with no parameters uses today's date as the default value:

```
type
  TDate = class
  public
    constructor Create; overload;
    constructor Create (Month, Day, Year: Integer); overload;

  constructor TDate.Create (Month, Day, Year: Integer);
  begin
```

```

    FDate := EncodeDate (Year, Month, Day);
end;

constructor TDate.Create;
begin
    FDate := Date;
end;

```

Having these two constructors makes it possible to define a new `TDate` object in two different ways:

```

var
    Day1, Day2: TDate;
begin
    Day1 := TDate.Create (2020, 12, 25);
    Day2 := TDate.Create; // today

```

This code is part of the `Dates4` application project.

The Complete TDate Class

Throughout this chapter, I've shown you bits and pieces of the source code for different versions of a `TDate` class. The first version was based on three integers to store the year, the month, and the day; a second version used a field of the `TDateTime` type provided by the RTL. Here is the complete interface portion of the unit that defines the `TDate` class:

```

unit Dates;

interface

type
    TDate = class
    private
        FDate: TDateTime;
    public
        constructor Create; overload;
        constructor Create (Month, Day, Year: Integer); overload;
        procedure SetValue (Month, Day, Year: Integer); overload;
        procedure SetValue (NewDate: TDateTime); overload;
        function LeapYear: Boolean;
        procedure Increase (NumberOfDays: Integer = 1);
        procedure Decrease (NumberOfDays: Integer = 1);
        function GetText: string;
    end;

```

The aim of the new methods, `Increase` and `Decrease` (which have a default value for their parameter), is quite easy to understand. If called with no parameter, they change the value of the date to the next or previous day. If a `NumberOfDays` parameter is part of the call, they add or subtract that number:

234 - 07: Objects

```
procedure TDate.Increase (NumberOfDays: Integer = 1);  
begin  
    FDate := FDate + NumberOfDays;  
end;
```

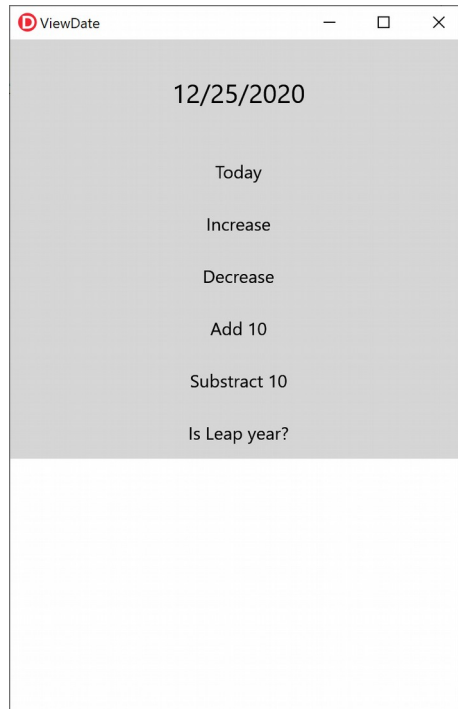
The method `GetText` returns a string with the formatted date, using the `DateToStr` function for the conversion:

```
function TDate.GetText: string;  
begin  
    GetText := DateToStr (FDate);  
end;
```

We've already seen most of the methods in the previous sections, so I won't provide the complete listing; you can find it in the code of the `ViewDate` application project I've written to test the class. The form is a little more complex than others in the book, and it has a caption to display a date and six buttons, which can be used to modify the value of the object. You can see the main form of the `ViewDate` application project at run time in Figure 7.3. To make the `Label` component look nice, I've given it a big font, made it as wide as the form, set its `Alignment` property to `taCenter`, and set its `AutoSize` property to `False`.

Figure 7.3:

The output of the
`ViewDate` application
at start-up



The start-up code of this program is in the `OnCreate` event handler of the form. In the corresponding method, we create an instance of the `TDate` class, initialize this object, and then show its textual description in the `Text` of the label, as shown in Figure 7.3.

```
procedure TDateForm.FormCreate(Sender: TObject);
begin
    ADay := TDate.Create;
    LabelDate.Text := ADay.GetText;
end;
```

`ADay` is a private field of the class of the form, `TDateForm`. By the way, the name for the class is automatically chosen by the development environment when you change the `Name` property of the form to `DateForm`.

The specific date object is created when the form is created (setting up the same relationship we saw earlier between the person class and its date sub-object) and is then destroyed along with the form:

```
procedure TDateForm.FormDestroy(Sender: TObject);
begin
    ADay.Free;
end;
```

When the user clicks one of the six buttons, we need to apply the corresponding method to the `ADay` object and then display the new value of the date in the label:

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
begin
    ADay.SetValue (Today);
    LabelDate.Text := ADay.GetText;
end;
```

An alternative way to write the last method is to destroy the current object and create a new one:

```
procedure TDateForm.BtnTodayClick(Sender: TObject);
begin
    ADay.Free;
    ADay := TDate.Create;
    LabelDate.Text := ADay.GetText;
end;
```

In this particular circumstance, this is not a very good approach (because creating a new object and destroying an existing one entails a lot of time overhead, when all we need is to change the object's value), but it allows me to show you a couple of Object Pascal techniques. The first thing to notice is that we destroy the previous object before assigning a new one. The assignment operation, in fact, replaces the reference, leaving the object in memory (even if no pointer is referring to it). When you assign an object to another object, the compiler simply copies the reference to the object in memory to the new object reference.

236 - 07: Objects

One side issue is how do you copy the data from one object to another. This case is very simple, because there is only one field and a method to initialize it. In general if you want to change the data inside an existing object, you have to copy each field, or provide a specific method to copy all of the internal data. Some classes have an `Assign` method, which does this *deep-copy* operation.

note To be more precise, in the runtime library all of the classes inheriting from `TPersistent` have the `Assign` method, but most of those inheriting from `TComponent` don't implement it, raising an exception when it is called. The reason lies in the streaming mechanism supported by the runtime libraries and the support for properties of `TPersistent` types, but this is way too complex to delve into at this point of the book.

Nested Types and Nested Constants

Object Pascal allows you to declare new classes in the interface section of a unit, allowing other units of the program to reference them, or in the implementation section, where they are accessible only from methods of other classes of the same unit or from global routines implemented in that unit after the class definition.

A more recent addition is the possibility to declare a class (or any other data type) within another class. As any other member of the class, the nested types can have a restricted visibility (say, `private` or `protected`). Relevant examples of nested types include enumerations used by the same class and implementation-support classes.

A related syntax allows you to define a nested constant, a constant value associated with the class (again usable only internally, if `private`, or usable by the rest of the program, if `public`). As an example, consider the following declaration of a nested class (extracted from the `NestedClass` unit of the `NestedTypes` application project):

```
type
  TOne = class
    private
      FSomeData: Integer;
    public
      // nested constant
      const Foo = 12;
      // nested type
      type
        TInside = class
          public
            procedure InsideHello;
          private
            FMsg: string;
```

```

        end;
    public
        procedure Hello;
    end;

procedure TOne.Hello;
var
    Ins: TInside;
begin
    Ins := TInside.Create;
    Ins.FMsg := 'hi';
    Ins.InsideHello;
    Show ('Constant is ' + IntToStr (Foo));
    Ins.Free;
end;

procedure TOne.TInside.InsideHello;
begin
    FMsg := 'new msg';
    Show ('internal call');
    if not Assigned (InsIns) then
        InsIns := TInsideInside.Create;
    InsIns.Two;
end;

procedure TOne.TInside.TInsideInside.Two;
begin
    Show ('this is a method of a nested/nested class');
end;

```

The nested class can be used directly within the class (as demonstrated in the listing) or outside the class (if it is declared in the public section), but with the fully qualified name `TOne.TInside`. The *full name* of the class is used also in the definition of the method of the nested class, in this case `TOne.TInside`. The hosting class can have a field of the nested class type immediately after you've declared the nested class (as you can see in the code of the NestedClass application project).

The class with the nested classes is used as follows:

```

var
    One: TOne;
begin
    One := TOne.Create;
    One.Hello;
    One.Free;

```

This produces the following output:

```

internal call
this is a method of a nested/nested class
constant is 12

```

How would you benefit from using a nested class in the Object Pascal language? The concept is commonly used in Java to implement event handler delegates and makes

238 - 07: Objects

sense in C# where you cannot hide a class inside a unit. In Object Pascal nested classes are the only way you can have a field of the type of another private class (or inner class) without adding it to the global name space, making it globally visible.

If the internal class is used only by a method, you can achieve the same effect by declaring the class within the implementation portion of the unit. But if the inner class is referenced in the interface section of the unit (for example because it is used for a field or a parameter), it must be declared in the same interface section and will end up being visible. The trick of declaring such a field of a generic or base type and then casting it to the specific (private) type is much less clean than using a nested class.

note In chapter 10 there is a practical example in which nested classes come in handy, namely implementing a custom iterator for a `for in` loop.

08: inheritance

If the key reason for writing classes is encapsulation, the key reason for using inheritance among classes is flexibility. Combine the two concepts and you can have data types you can use and are not going to change with the ability to create modified versions of those types, in what was originally known as the “*open-closed principle*”:

“Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.” (Bertrand Meyer, Object-Oriented Software Construction, 1988)

Now it is true that inheritance is a very strong binding leading to tight coupled code, but it is also true it offers great power to the developer (and, yes, also the more responsibility that comes with it).

Rather than opening up a debate on this feature, however, my goal here is to describe you how type inheritance works and specifically how it works in the Object Pascal language.

Inheriting from Existing Types

We often need to use a slightly different version of an existing class that we have written or that someone has given to us.

For example, you might need to add a new method or slightly change an existing one. You can do this easily by modifying the original code, unless you want to be able to use the two different versions of the class in different circumstances. Also, if the class was originally written by someone else (and you have found it in a library), you might want to keep your changes separate.

A typical old-school alternative for having two similar versions of a class is to make a copy of the original type definition, change its code to support the new features, and give a new name to the resulting class. This might work, but it also might create problems: in duplicating the code you also duplicate the bugs; when a bug is fixed in one of the copies of the code, you'd have to remember applying the fix to the other copy; and if you want to add a new feature, you'll need to add it two or more times, depending on the number of copies of the original code you've made over time. Even if this might not slow down you when you write the code the first time, this is a disaster for software maintenance. Moreover, this approach results in two completely different data types, so the compiler cannot help you take advantage of the similarities between the two types.

To solve these kinds of problems in expressing similarities between classes, Object Pascal allows you to define a new class directly from an existing one. This technique is known as *inheritance* (or *subclassing*, or *type derivation*) and is one of the fundamental elements of object-oriented programming languages.

To inherit from an existing class, you only need to indicate that class at the beginning of the declaration of the subclass. For example, this is done automatically each time you create a new form:

```
type
  TForm1 = class(TForm)

end;
```

This simple definition indicates that the `TForm1` class inherits all the methods, fields, properties, and events of the `TForm` class. You can apply any public method of the `TForm` class to an object of the `TForm1` type. `TForm`, in turn, inherits some of its methods from another class, and so on, up to the `TObject` class (which is the base class of all classes).

By comparison C++, C# and Java would use something like:

```
class Form1 : TForm
```

```
{
  ...
}
```

As a simple example of inheritance, we can change the `viewDate` application project of the last chapter slightly, deriving a new class from `TDate` and modifying one of its functions, `GetText`. You can find this code in the `DATES.PAS` file of the `DerivedDates` application project.

```
type
  TNewDate = class (TDate)
  public
    function GetText: string;
  end;
```

In this example, `TNewDate` is derived from `TDate`. It is common to say that `TDate` is an *ancestor* class or *base* class or *parent* class of `TNewDate` and that `TNewDate` is a *subclass*, *descendant* class, or *child* class of `TDate`.

To implement the new version of the `GetText` function, I used the `FormatDateTime` function, which uses (among other features) the predefined month names. Here is the `GetText` method, where `'dddddd'` stands for the long data format:

```
function TNewDate.GetText: string;
begin
  Result := FormatDateTime ('dddddd', FDate);
end;
```

Once we have defined the new class, we need to use this new data type in the code of the form of the `DerivedDates` project. Simply define the `ADay` object of type `TNewDate`, and call its constructor in the `FormCreate` method:

```
type
  TDateForm = class(TForm)
  ...
  private
    ADay: TNewDate; // updated declaration
  end;

procedure TDateForm.FormCreate(Sender: TObject);
begin
  ADay := TNewDate.Create; // updated line
  DateLabel.text := TheDay.GetText;
end;
```

Without any other changes, the new application will work properly.

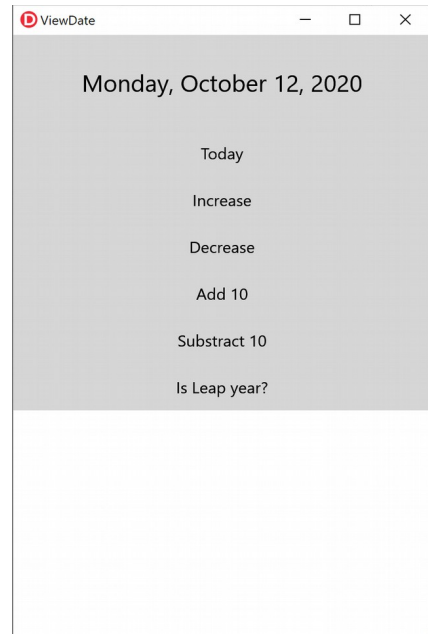
The `TNewDate` class inherits the methods to increase the date, add a number of days, and so on. In addition, the older code calling these methods still works. Actually, to call the new version of the `GetText` method, we don't need to change the source code! The Object Pascal compiler will automatically bind that call to a new method.

242 - 08: Inheritance

The source code of all the other event handlers remains exactly the same, although its meaning changes considerably, as the new output demonstrates (see Figure 8.1).

Figure 8.1:

The output of the
DerivedDates program,
with the name of the
month and of the day
depending on Windows
regional settings



A Common Base Class

We have seen that if you can inherit from a given base class by writing:

```
type
  TNewDate = class (TDate)
    ...
  end;
```

But what happens if you omit a base class and write:

```
type
  TNewDate = class
    ...
  end;
```

In this case your class inherits from a base class, called `TObject`. In other words Object Pascal has a single-rooted class hierarchy, in which all classes directly or

indirectly inherit from a common ancestor class. The most commonly used methods of `TObject` are `Create`, `Free`, and `Destroy`; but there are many others I'll use throughout the book. A complete description of this fundamental class (that could be considered both part of the language and also part of the runtime library) with a reference to all of its methods is available in Chapter 17.

note The concept of a common ancestor class is present also in the C# and Java languages, where this is simply called `Object`. The C++ language, on the other hand, hasn't got such an idea, and a C++ program generally has multiple independent class hierarchies.

Protected Fields and Encapsulation

The code of the `GetText` method of the `TNewDate` class compiles only if it is written in the same unit as the `TDate` class. In fact, it accesses the `FDate` private field of the ancestor class. If we want to place the descendant class in a new unit, we must either declare the `FDate` field as `protected` (or `strict protected`) or add a simple, possibly protected method in the ancestor class to read the value of the private field.

Many developers believe that the first solution is always the best, because declaring most of the fields as protected will make a class more extensible and will make it easier to write subclasses. However, this violates the idea of encapsulation. In a large hierarchy of classes, changing the definition of some protected fields of the base classes becomes as difficult as changing some global data structures. If ten derived classes are accessing this data, changing its definition means potentially modifying the code in each of the ten classes.

In other words, flexibility, extension, and encapsulation often become conflicting objectives. When this happens, you should try to favor encapsulation. If you can do so without sacrificing flexibility, that will be even better. Often this intermediate solution can be obtained by using a virtual method, a topic I'll discuss in detail below in the section "Late Binding and Polymorphism." If you choose not to use encapsulation in order to obtain faster coding of the subclasses, then your design might not follow the object-oriented principles.

Remember also that protected fields share the same access rules of private ones, so that any other class in the same unit can always access protected members of other classes. As mentioned in the previous chapter, you can use stronger encapsulation by using the `strict protected` access specifier.

Using the “Protected Hack”

If you are new to Object Pascal and to OOP, this is a rather advanced section you might want to skip the first time you are reading this book, as it might be quite confusing.

Given how unit protection works, even protected members of base classes of classes declared in the current unit can be directly accessed, unless you use the `strict protected` keyword. This is the rationale behind what is generally called “*the protected hack*”, that is the ability to define a derived class identical to its base class for the only purpose of gaining access at the protected member of the base class. Here is how it works.

We’ve seen that the private and protected data of a class is accessible to any functions or methods that appear *in the same unit as the class*. For example, consider this simple class (part of the Protection application project):

```
type
  TTest = class
    protected
      ProtectedData: Integer;
    public
      PublicData: Integer;
      function GetValue: string;
    end;
```

The `GetValue` method simply returns a string with the two integer values:

```
function TTest.GetValue: string;
begin
  Result := Format ('Public: %d, Protected: %d',
    [PublicData, ProtectedData]);
end;
```

Once you place this class in its own unit, you won’t be able to access its protected portion from other units directly. Accordingly, if you write the following code,

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Obj: TTest;
begin
  Obj := TTest.Create;
  Obj.PublicData := 10;
  Obj.ProtectedData := 20; // won't compile
  Show (Obj.GetValue);
  Obj.Free;
end;
```

the compiler will issue an error message, *Undeclared identifier: “ProtectedData.”* At this point, you might think there is no way to access the protected data of a class defined in a different unit. However, there is a way around it.

Consider what happens if you create an apparently useless derived class, such as:

```
type
  TFake = class (TTest);
```

Now, in the same unit where you have declared it, you can call any protected method of the `TFake` class. In fact you can call protected methods of a class declared in the same unit.

How does this helps using an object of class `TTest`, though? Considering that the two classes share the same exact memory layout (as there are no differences) you can force the compiler to treat an object of a class like one of the other, with what is generally a type-unsafe cast:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Obj: TTest;
begin
  Obj := TTest.Create;
  Obj.PublicData := 10;
  TFake (Obj).ProtectedData := 20; // compiles!
  Show (Obj.GetValue);
  Obj.Free;
end;
```

This code compiles and works properly, as you can see by running the `Protection` application project. Again, the reason is that the `TFake` class automatically inherits the protected fields of the `TTest` base class, and because the `TFake` class is in the same unit as the code that tries to access the data in the inherited fields, the protected data is accessible.

Now that I've shown you how to do this, I must warn you that violating the class-protection mechanism this way is likely to cause errors in your program (from accessing data that you really shouldn't), and it runs counter to good OOP technique. However, there are rare times when using this technique is the best solution, as you'll see by looking at the library source code and the code of many components.

Overall, this technique is a *hack* and it should be avoided whenever possible, although it can be considered to all effects as part of the language specification and is available on all platforms and in all present and past versions of Object Pascal.

From Inheritance to Polymorphism

Inheritance is a nice technique in terms of letting you avoid code duplication and share code methods among different classes. Its true power, however, comes from the ability to handle objects of different classes in a uniform manner, something often indicated in object-oriented programming languages by the term *polymorphism* or referenced as *late binding*.

There are several elements we have to explore to fully understand this feature: type compatibility among derived classes, virtual methods, and more, as covered in the next few sections.

Inheritance and Type Compatibility

As we have seen to some extent, Object Pascal is a strictly typed language. This means that you cannot, for example, assign an integer value to a `Boolean` variable, at least not without an explicit typecast. The basic rule is that two values are type-compatible only if they are of the same data type, or (to be more precise) if their data type has the same name and their definition comes from the same unit.

There is an important exception to this rule in the case of class types. If you declare a class, such as `TAnimal`, and derive from it a new class, say `TDog`, you can then assign an object of type `TDog` to a variable of type `TAnimal`. That is because a dog is an animal! So, although this might surprise you, the following constructor calls are both legal:

```
var
  MyAnimal1, MyAnimal2: TAnimal;
begin
  MyAnimal1 := TAnimal.Create;
  MyAnimal2 := TDog.Create;
```

In more precise terms, you can use an object of a descendant class any time an object of an ancestor class is expected. However, the reverse is not legal; you cannot use an object of an ancestor class when an object of a descendant class is expected. To simplify the explanation, here it is again in code terms:

```
MyAnimal := MyDog; // This is OK
MyDog := MyAnimal; // This is an error!!!
```

In fact, while we can always say that a dog is an animal, we cannot assume that any given animal is a dog. This might be true at times, but not always. This is quite logical, and the language type compatibility rules follow this same logic.

Before we look at the implications of this important feature of the language, you can try out the `Animals1` application project, which defines the two simple `TAnimal` and `TDog` classes, inheriting one from the other:

```
type
  TAnimal = class
    public
      constructor Create;
      function GetKind: string;
    private
      FKind: string;
    end;

  TDog = class (TAnimal)
    public
      constructor Create;
    end;
```

The two `Create` methods simply set the value of `FKind`, which is returned by the `GetKind` function.

The form of this example, shown in Figure 8.2, has two radio buttons (hosted by a panel) to pick an object of one or the other class. This object is stored in the private field `MyAnimal` of type `TAnimal`. An instance of this class is created and initialized when the form is created and re-created each time one of the radio buttons is selected (here I'm showing only the code of the second radio button):

```
procedure TFormAnimals.FormCreate(Sender: TObject);
begin
  MyAnimal := TAnimal.Create;
end;

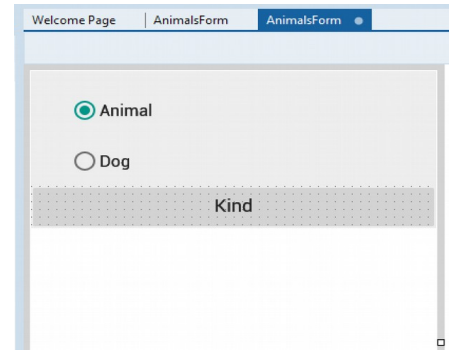
procedure TFormAnimals.RadioButton2Change(Sender: TObject);
begin
  MyAnimal.Free;
  MyAnimal := TDog.Create;
end;
```

Finally, the `Kind` button calls the `GetKind` method for the current animal and displays the result in the memo covering the bottom part of the form:

```
procedure TFormAnimals.BtnKindClick(Sender: TObject);
begin
  Show(MyAnimal.GetKind);
end;
```


Figure 8.2:

The form of the Animals1 application project in the development environment



Late Binding and Polymorphism

Object Pascal functions and procedures are usually based on *static binding*, which is also called *early binding*. This means that a method call is resolved by the compiler or the linker, which replaces the request with a call to the specific memory location where the compiled function or procedure resides. (This is also known as the *address* of the function.) Object-oriented programming languages allow the use of another form of binding, known as *dynamic binding*, or *late binding*. In this case, the actual address of the method to be called is determined at run time based on the type of the instance used to make the call.

The advantage of this technique is known as *polymorphism*. Polymorphism means you can write a call to a method, applying it to a variable, but which method Delphi actually calls depends on the type of the object the variable relates to. Delphi cannot determine until run time the actual class of the object the variable refers to, simply because of the type-compatibility rule discussed in the previous section.

note Object Pascal methods default to early binding, like C++ and C#. One of the reasons is this is more efficient. Java, instead, defaults to late binding (and offers ways to indicate to the compiler it can optimize a method using early binding).

Suppose that a class and its subclass (let's say `TAnimal` and `TDog`, again) both define a method, and this method has late binding. Now you can apply this method to a generic variable, such as `MyAnimal`, which at run time can refer either to an object of class `TAnimal` or to an object of class `TDog`. The actual method to call is determined at run time, depending on the class of the current object.

The `Animals2` application project extends the `Animals1` project to demonstrate this technique. In the new version, the `TAnimal` and the `TDog` classes have a new method: `Voice`, which means to output the sound made by the selected animal, both as text and as sound. This method is defined as `virtual` in the `TAnimal` class and is later overridden when we define the `TDog` class, by the use of the `virtual` and `override` keywords:

```
type
  TAnimal = class
    public
      function Voice: string; virtual;

  TDog = class (TAnimal)
    public
      function Voice: string; override;
```

Of course, the two methods also need to be implemented. Here is a simple approach:

```
function TAnimal.Voice: string;
begin
  Result := 'AnimalVoice';
end;

function TDog.Voice: string;
begin
  Result := 'ArfArf';
end;
```

Now what is the effect of the call `MyAnimal.Voice`? It depends. If the `MyAnimal` variable currently refers to an object of the `TAnimal` class, it will call the method `TAnimal.Voice`. If it refers to an object of the `TDog` class, it will call the method `TDog.Voice` instead. This happens only because the function is `virtual`.

The call to `MyAnimal.Voice` will work for an object that is an instance of any descendant of the `TAnimal` class, even classes that are defined after this method call or outside its scope. The compiler doesn't need to know about all the descendants in order to make the call compatible with them; only the ancestor class is needed. In other words, this call to `MyAnimal.Voice` is compatible with all future `TAnimal` subclasses.

This is the key technical reason why object-oriented programming languages favor re-usability. You can write code that uses classes within a hierarchy without any knowledge of the specific classes that are part of that hierarchy. In other words, the hierarchy—and the program—is still extensible, even when you've written thousands of lines of code using it. Of course, there is one condition—the ancestor classes of the hierarchy need to be designed very carefully.

250 - 08: Inheritance

The `Animals2` application project demonstrates the use of these new classes and has a form similar to that of the previous example. This code is executed by clicking on the button, showing the output and also producing some sound:

```
begin
  Show (MyAnimal.Voice);
  MediaPlayer1.FileName := SoundsFolder + MyAnimal.Voice + '.wav';
  MediaPlayer1.Play;
end;
```

note The application uses a `MediaPlayer` component to play one of the two sound files that come with the application (the sound files are named after the actual sounds, that is the values returned by the `voice` method). A rather random noise for the generic animal, and some barking for the dog. Now the code works easily on Windows, as long as the files are in the proper folder, but it requires some effort for the deployment on mobile platforms. Take a look at the actual demo to see how the deployment and the folders are structured.

Overriding, Redefining, and Reintroducing Methods

As we have just seen, to override a late-bound method in a descendant class, you need to use the `override` keyword. Note that this can take place only if the method was defined as `virtual` in the ancestor class. Otherwise, if it was a static method, there is no way to activate late binding, other than by changing the code of the ancestor class.

note You might remember I used the same keyword also in the last chapter to override the `Destroy` default destructor, inherited from the base `TObject` class.

The rules are simple: A method defined as static remains static in every subclass, unless you hide it with a new virtual method having the same name. A method defined as `virtual` remains late-bound in every subclass. There is no way to change this, because of the way the compiler generates different code for late-bound methods.

To redefine a static method, you simply add a method to a subclass having the same parameters or different parameters than the original one, without any further specifications. To override a `virtual` method, you must specify the same parameters and use the `override` keyword:

```
type
  TMyClass = class
    procedure One; virtual;
```

```

procedure Two; // static method
end;

TMySubClass = class (MyClass)
  procedure One; override;
  procedure Two;
end;

```

The redefined method, `Two`, has no late binding. So when you apply it to a variable of the base class, it calls the base class method no matter what (that is, even if the variable is referring to an object of the derived class, that has a different version for that method).

There are typically two ways to override a method. One is to replace the method of the ancestor class with brand a new version. The other is to add some more code to the existing method. This second approach can be accomplished by using the `inherited` keyword to call the same method of the ancestor class. For example, you can write

```

procedure TMySubClass.One;
begin
  // new code
  ...
  // call inherited procedure TMyClass.One
  inherited One;
end;

```

You might wonder why you need to use the `override` keyword. In other languages, when you redefine a virtual method in a subclass, you automatically override the original one. However, having a specific keyword allows the compiler to check the correspondence between the name of the method in the ancestor class and name of the method in the subclass (misspelling a redefined function is a common error in some other OOP languages), check that the method was virtual in the ancestor class, and so on.

note There is another popular OOP language that has the same `override` keyword, C#. This is not surprising, given the fact the languages share a common designer. Anders Hejlsberg has some lengthy articles explaining why the `override` keyword is a fundamental versioning tool for designing libraries, as you can read at <http://www.artima.com/intv/nonvirtual.html>. More recently, Apple's Swift language has also adopted the `override` keyword to modify methods in derived classes.

Another advantage of this keyword is that if you define a static method in any class inherited by a class of the library, there will be no problem, even if the library is updated with a new virtual method having the same name as a method you've defined. Because your method is not marked by the `override` keyword, it will be

252 - 08: Inheritance

considered a separate method and not a new version of the one added to the library (something that would probably break your existing code).

The support for overloading adds some further complexity to this picture. A subclass can provide a new version of a method using the `overload` keyword. If the method has different parameters than the version in the base class, it becomes effectively an overloaded method; otherwise it replaces the base class method. Here is an example:

```
type
  TMyClass = class
    procedure One;
  end;

  TMySubClass = class (TMyClass)
    procedure One (S: string); overload;
  end;
```

Notice that the method doesn't need to be marked as `overload` in the base class. However, if the method in the base class is virtual, the compiler issues the warning *Method 'One' hides virtual method of base type 'TMyClass.'*

To avoid this message from the compiler and to instruct the compiler more precisely on your intentions, you can use the specific `reintroduce` directive:

```
type
  TMyClass = class
    procedure One; virtual;
  end;

  TMySubClass = class (TMyClass)
    procedure One (S: string); reintroduce; overload;
  end;
```

You can find this code in the `ReintroduceTest` application project and experiment with it further.

note A scenario in which the `reintroduce` keyword is used is when you want to add a custom `Create` constructor to a component class, that already inherits a virtual `Create` constructor from the `TComponent` base class.

Inheritance and Constructors

As we have seen, you can use the `inherited` keyword to invoke same name method (or also a different method) in a method of a derived class. The same is also true for constructors. While in other languages like C++, C# or Java, the call to the base class constructor is implicit and compulsory (when you have to pass parameters to

the base class constructor), in Object Pascal calling a base class constructor is not strictly required.

In most cases, however, manually calling the base class constructor is extremely important. This is the case, for example, for any component class, as the component initialization is actually done at the `TComponent` class level:

```
constructor TMyComponent.Create (Owner: TComponent);
begin
    inherited Create (Owner);
    // specific code...
end;
```

This is particularly important because for components `Create` is a virtual method. Similarly for all classes, the `Destroy` destructor is a virtual method and you should remember calling `inherited` in it.

One question remains: If you are creating a class, which only inherits from `TObject`, in its constructors do you need to call the base `TObject.Create` constructor? From a technical point of view, the answer is “no” given that constructor is empty. However, I consider it a good habit to always call the base class constructor, no matter what. If you are a performance maniac, however, I’ll concede this can needlessly slow down your code... by a completely unnoticeable fraction of microsecond.

Jokes aside, there are good reasons for both approaches, but particularly for a beginner with the language I recommend always calling the base class constructor as good programming habit, promoting safer coding.

Virtual versus Dynamic Methods

In Object Pascal, there are two different ways to activate late binding. You can declare a method as `virtual`, as we have seen before, or declare it as `dynamic`. The syntax of these two keywords is exactly the same, and the result of their use is also the same. What is different is the internal mechanism used by the compiler to implement late binding.

Virtual methods are based on a *virtual method table* (or VMT, but colloquially also known as a *vtable*). A virtual method table is an array of method addresses. For a call to a virtual method, the compiler generates code to jump to an address stored in the *n*th slot in the object’s virtual method table.

Virtual method tables allow fast execution of the method calls. Their main drawback is that they require an entry for each virtual method for each descendant class, even if the method is not overridden in the subclass. At times, this has the effect of propagating virtual method table entries throughout a class hierarchy (even for methods

254 - 08: Inheritance

that aren't redefined). This might require a lot of memory just to store the same method address a number of times.

Dynamic method calls, on the other hand, are dispatched using a unique number indicating the method. The search for the corresponding function is generally slower than the simple one-step table lookup for virtual methods. The advantage is that dynamic method entries only propagate in descendants when the descendants override the method. For large or deep object hierarchies, using dynamic methods instead of virtual methods can result in significant memory savings with only a minimal speed penalty.

From a programmer's perspective, the difference between these two approaches lies only in a different internal representation and slightly different speed or memory usage. Apart from this, virtual and dynamic methods are the same.

Now having explained the difference between these two models, it is important to underline that in the largest number of cases, application developers use `virtual` rather than `dynamic`.

Message Handlers on Windows

When you are building applications for Windows, a special purpose late-bound method can be used to handle a Windows system message. For this purpose Object Pascal provides yet another directive, `message`, to define message-handling methods, which must be procedures with a single `var` parameter of the proper type. The `message` directive is followed by the number of the Windows message the method wants to handle. For example, the following code allows you to handle a user-defined message, with the numeric value indicated by the `WM_USER` Windows constant:

```
type
  TForm1 = class(TForm)
  ...
  procedure WmUser (var Msg: TMessage); message WM_USER;
end;
```

The name of the procedure and the actual type of the parameters are up to you, as long as the physical data structure matches the Windows message structure. The units used to interface with the Windows API include a number of predefined record types for the various Windows messages. This technique can be extremely useful for veteran Windows programmers, who know all about Windows messages and API functions, but it absolutely not compatible with other operating systems (like macOS, iOS, and Android).

Abstracting Methods and Classes

When you are creating a hierarchy of classes, at times it is difficult to determine which is the base class, given it might not represent an actual entity, but only be used to hold some shared behavior. An example would be an animal base class for something like a cat or a dog class. Such a class for which you are not expected to create any object is often indicated as an *abstract* class, because it has no concrete and complete implementation. An abstract class can have abstract methods, methods that don't have an actual implementation.

Abstract Methods

The `abstract` keyword is used to declare virtual methods that will be defined only in subclasses of the current class. The `abstract` directive fully defines the method; it is not a forward declaration. If you try to provide a definition for the method, the compiler will complain.

In Object Pascal, you can create instances of classes that have abstract methods. However, when you try to do so, the compiler issues the warning message: *Constructing instance of <class name> containing abstract methods*. If you happen to call an abstract method at run time, Delphi will raise a specific runtime exception.

note C++, Java, and other languages use a more strict approach: in these languages, you cannot create instances of abstract classes.

You might wonder why you would want to use abstract methods. The reason lies in the use of polymorphism. If class `TAnimal` has the `virtual` abstract method `voice`, every subclass can redefine it. The advantage is that you can now use the generic `MyAnimal` object to refer to each animal defined by a subclass and invoke this method. If this method was not present in the interface of the `TAnimal` class, the call would not have been allowed by the compiler, which performs static type checking. Using a generic `MyAnimal` object, you can call only the method defined by its own class, `TAnimal`.

You cannot call methods provided by subclasses, unless the parent class has at least the declaration of this method—in the form of an abstract method. The next application project, `Animals3`, demonstrates the use of abstract methods and the abstract call error. Here are the interfaces of the classes of this new example:

■ **type**

256 - 08: Inheritance

```
TAnimal = class
public
  constructor Create;
  function GetKind: string;
  function Voice: string; virtual; abstract;
private
  FKind: string;
end;

TDog = class (TAnimal)
public
  constructor Create;
  function Voice: string; override;
  function Eat: string; virtual;
end;

TCat = class (TAnimal)
public
  constructor Create;
  function Voice: string; override;
  function Eat: string; virtual;
end;
```

The most interesting portion is the definition of the class `TAnimal`, which includes a virtual abstract method: `Voice`. It is also important to notice that each derived class overrides this definition and adds a new virtual method, `Eat`. What are the implications of these two different approaches? To call the `Voice` function, we can simply write the same code as in the previous version of the program:

```
| Show (MyAnimal.Voice);
```

How can we call the `Eat` method? We cannot apply it to an object of the `TAnimal` class. The statement

```
| Show (MyAnimal.Eat);
```

generates the compiler error *Field identifier expected*.

To solve this problem, you can use a dynamic and safe type cast to treat the `TAnimal` object as a `TCat` or as a `TDog` object, but this would be a very cumbersome and error-prone approach:

```
begin
  if MyAnimal is TDog then
    Show (TDog(MyAnimal).Eat)
  else if MyAnimal is TCat then
    Show (TCat(MyAnimal).Eat);
```

This code will be explained later in the section “Safe Type Cast Operators”. Adding the virtual method definition to the `TAnimal` class is a typical solution to the problem, and the presence of the `abstract` keyword favors this choice. The code above looks ugly, and avoiding such a code is precisely the reason for using polymorphism.

Finally notice that when a class has an abstract method, it is often considered to be an abstract class. However you can also specifically mark a class with the `abstract` directive (and it will be considered an abstract class even if it has no abstract methods). Again, in Object Pascal this won't prevent you from creating an instance of the class, so in this language the usefulness of an abstract class declaration is quite limited.

Sealed Classes and Final Methods

As I mentioned, Java has a very dynamic approach with late binding (or virtual functions) being the default. For this reason the language introduced concepts like classes you cannot inherit from (*sealed*) and methods you cannot override in derived classes (*final methods*, or non-virtual methods).

Sealed classes are classes you cannot further inherit from. This might make sense if you are distributing components (without the source code) or runtime packages and you want to limit the ability of other developers to modify your code. One of the original goals was also to increase runtime security, something you won't generally need in a fully compiled language like Object Pascal.

Final methods are virtual methods you cannot further override in inherited classes. Again, while they do make sense in Java (where all methods are virtual by default and final methods are significantly optimized) they were adopted in C# where virtual functions are explicitly marked and are much less important. Similarly, they were added to Object Pascal, where they are rarely used.

In terms of syntax, this is the code of a sealed class:

```
type
  TDeriv1 = class sealed (TBase)
    procedure A; override;
  end;
```

Trying to inherit from it causes the error, “*Cannot extend sealed class TDeriv1*”. This is the syntax of a final method:

```
type
  TDeriv2 = class (TBase)
    procedure A; override; final;
  end;
```

Inheriting from this class and overriding the `A` method causes the compiler error, “*Cannot override a final method*”.

Safe Type Cast Operators

As we have seen earlier, the language type compatibility rule for descendant classes allows you to use a descendant class where an ancestor class is expected. As I mentioned, the reverse is not possible.

Now suppose that the `TDog` class has an `Eat` method, which is not present in the `TAnimal` class. If the variable `MyAnimal` refers to a dog, you might want to be able to call the function. But if you try, and the variable is referring to another class, the result is an error. By making an explicit typecast, we could cause a nasty run-time error (or worse, a subtle memory overwrite problem), because the compiler cannot determine whether the type of the object is correct and the methods we are calling actually exist.

To solve the problem, we can use techniques based on run-time type information. Essentially, because each object at run time “knows” its type and its parent class. We can ask for this information with the `is` operator or using some of the methods of the `TObject` class. The parameters of the `is` operator are an object and a class type, and the return value is a Boolean:

```
if MyAnimal is TDog then
    ...
```

The `is` expression evaluates as `True` only if the `MyAnimal` object is currently referring to an object of class `TDog` or a type descendant from and compatible with `TDog`. This means that if you test whether a `TDog` object stored in a `TAnimal` variable is really a `TDog` object, the test will succeed. In other words, this expression evaluates as `True` if you can safely assign the object (`MyAnimal`) to a variable of the data type (`TDog`).

note The actual implementation of the `is` operator is provided by the `InheritsFrom` method of the `TObject` class. So you could write the same expression as `MyAnimal.InheritsFrom(TDog)`. The reason to use this method directly comes from the fact that it can also be applied to class references and other special purpose types than don't support the `is` operator.

Now that you know for sure that the animal is a dog, you can use a direct type cast (that would in general be unsafe) by writing the following code:

```
if MyAnimal is TDog then
begin
    MyDog := TDog (MyAnimal);
    Text := MyDog.Eat;
end;
```

This same operation can be accomplished directly by another related type cast operator, `as`, which converts the object only if the requested class is compatible with the

actual one. The parameters of the `as` operator are an object and a class type, and the result is an object “converted” to the new class type. We can write the following snippet:

```
MyDog := MyAnimal as TDog;
Text := MyDog.Eat;
```

If we only want to call the `Eat` function, we might also use an even shorter notation:

```
(MyAnimal as TDog).Eat;
```

The result of this expression is an object of the `TDog` class data type, so you can apply to it any method of that class. The difference between the traditional cast and the use of the `as` cast is that the second one checks the actual type of the object and raises an exception if the type is not compatible with the type you are trying to cast it to. The exception raised is `EInvalidCast` (exceptions are described in the next chapter).

warning By contrast, in the C# language the `as` expression will return `nil` if the object is not type-compatible, while the direct type cast will raise an exception. So basically the two operations are reversed compared to Object Pascal.

To avoid this exception, use the `is` operator and, if it succeeds, make a plain type-cast (in fact there is no reason to use `is` and `as` in sequence, doing the type check twice – although you'll often see the combined use of `is` and `as`):

```
if MyAnimal is TDog then
  TDog(MyAnimal).Eat;
```

Both type cast operators are very useful in Object Pascal because you often want to write generic code that can be used with a number of components of the same type or even of different types. For example, when a component is passed as a parameter to an event-response method, a generic data type is used (`TObject`), so you often need to cast it back to the original component type:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if Sender is TButton then
    ...
end;
```

This is a common technique I'll use it in some later examples (events are introduced in Chapter 10).

The two type cast operators, `is` and `as`, are extremely powerful, and you might be tempted to consider them as standard programming constructs. Although they are indeed powerful, you should probably limit their use to special cases. When you need to solve a complex problem involving several classes, try using polymorphism

first. Only in special cases, where polymorphism alone cannot be applied, should you try using the type conversion operators to complement it.

note The use of the type cast operators has a slight negative impact on performance, because it must walk the hierarchy of classes to see whether the typecast is correct. As we have seen, virtual method calls just require a memory lookup, which is much faster.

Visual Form Inheritance

Inheritance is not only used in library classes or for the classes you write, but it's quite pervasive of the entire development environment based around Object Pascal. As we have seen, when you create a form in the IDE, this is an instance of a class that inherits from `TForm`. So any visual application has a structure based on inheritance, even in cases where you end up writing most your code in simple event handlers.

What is less known, though, even by more experienced developers, is that you can inherit a new form from one you've already created, a feature generally called *visual form inheritance* (and something quite peculiar to Object Pascal development environment).

The interesting element here is you can visually see the power of inheritance in action, and directly figure out its rules! In this useful also in practice? Well, it mostly depends on the kind of application you are building. If it has a number of forms, some of which are very similar to each other or simply include common elements, then you can place the common components and the common event handlers in the base form and add the specific behavior and components to the subclasses. Another common scenario is to use visual form inheritance to customize some of the forms of an applications for specific companies, without duplicating any source code (which is the core reason for using inheritance in the first place).

You can also use visual form inheritance to customize an application for different operating systems and form factors (phone to tablets, for example), without duplicating any source code or form definition code; just inherit the specific versions for a client from the standard forms. Remember that the main advantage of visual inheritance is that you can later change the original form and automatically update all the derived forms. This is a well-known advantage of inheritance in object-oriented programming languages. But there is a beneficial side effect: polymorphism.

You can add a virtual method to a base form and override it in a subclassed form. Then you can refer to both forms and call this method for each of them.

note Another approach in building forms with the same elements is to rely on frames, that is on visual composition of form panels. In both cases at design time you can work on two versions of a form. However, in visual form inheritance, you are defining two different classes (parent and derived), whereas with frames, you work on a frame class and an instance of that frame hosted by a form.

Inheriting From a Base Form

The rules governing visual form inheritance are quite simple, once you have a clear idea of what inheritance is. Basically, a subclass form has the same components as the parent form as well as some new components. You cannot remove a component of the base class, although (if it is a visual control) you can make it invisible. What's important is that you can easily change properties of the components you inherit.

Notice that if you change a property of a component in the inherited form, any modification of the same property in the parent form will have no effect. Changing other properties of the component will affect the inherited versions, as well. You can resynchronize the two property values by using the Revert to Inherited local menu command of the Object Inspector. The same thing is accomplished by setting the two properties to the same value and recompiling the code. After modifying multiple properties, you can resynchronize them all to the base version by applying the Revert to Inherited command of the component's local menu.

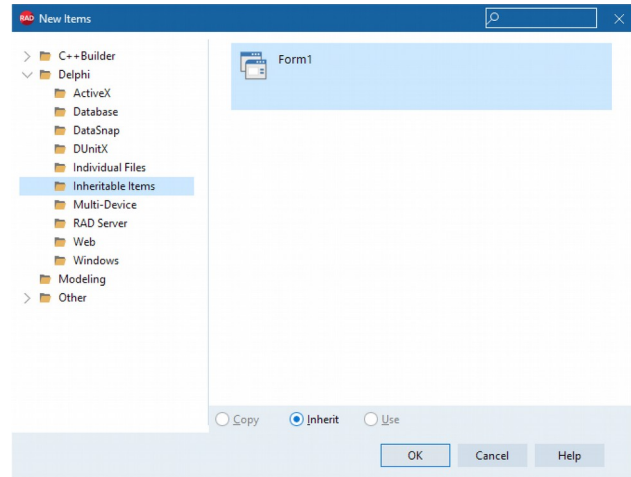
Besides inheriting components, the new form inherits all the methods of the base form, including the event handlers. You can add new handlers in the inherited form and also override existing handlers.

To demonstrate how visual form inheritance works, I've built a very simple example, called `visualInheritTest`. I'll describe step-by-step how to build it. First, start a new multi-device project, select a blank project, and add two buttons to its main form. Then select File ➤ New ➤ Others, and choose the "Inheritable Items" page in the New Items dialog box (see Figure 8.3). Here you can choose the form from which you want to inherit.

262 - 08: Inheritance

Figure 8.3:

The New Items dialog box allows you to create an inherited form.



The new form has the same two buttons. Here is the initial textual description of the new form:

```
inherited Form2: TForm2
  Caption = 'Form2'
  ...
end
```

And here is its initial class declaration, where you can see that the base class is not the usual `TForm` but the actual base class form:

```
type
  TForm2 = class(TForm1)
  private
    { Private declarations }
  public
    { Public declarations }
  end;
```

Notice the presence of the `inherited` keyword in the textual description; also notice that the form indeed has some components, although they are defined in the base class form. If you change the caption of one of the buttons and add a new button the textual description will change accordingly:

```
inherited Form2: TForm2
  Caption = 'Form2'
  ...
  inherited Button1: TButton
    Text = 'Hide Form'
  end
  object Button3: TButton
    ...
```

```

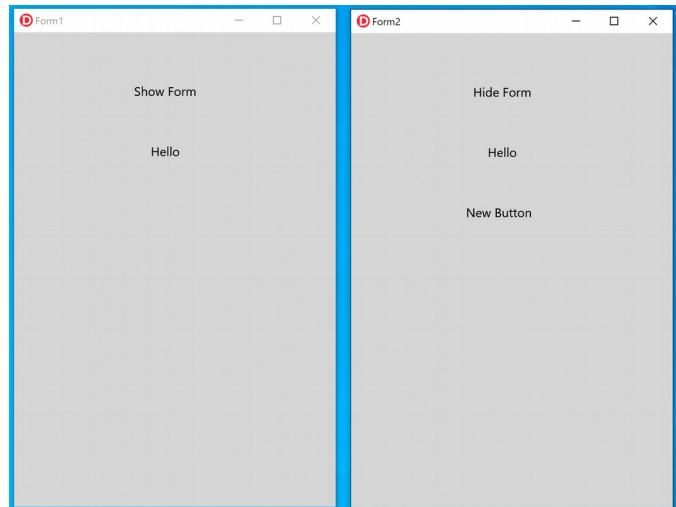
        Text = 'New Button'
        OnClick = Button3Click
    end
end

```

Only the properties with a different value are listed, because the others are simply inherited as they are.

Figure 8.4:

The two forms of the VirtualInheritTest example at run time



Each of the buttons of the first form has an `OnClick` handler, with simple code. The first button shows the second form calling its `Show` method; the second button a simple message.

What happens in the inherited form? First we should change the behavior of the `Show` button to implement it as a `Hide` button. This implies not executing the base class event handler (so I've commented out the default `inherited` call). For the `Hello` button, instead, we can add a second message to the one displayed by the base class, by leaving the `inherited` call:

```

procedure TForm2.Button1Click(Sender: TObject);
begin
    // inherited;
    Hide;
end;

procedure TForm2.Button2Click(Sender: TObject);
begin
    inherited;
    ShowMessage ('Hello from Form2');
end;

```


264 - o8: Inheritance

Remember that differently from an inherited method, that can use the `inherited` keyword to call the base class method with the same name, in an event handler the `inherited` keyword stands for a call to the corresponding event handler of the base form (regardless of the event handler method name).

Of course, you can also consider each method of the base form as a method of your form, and call them freely. This example allows you to explore some features of visual form inheritance, but to see its true power you'll need to look at more complex real-world examples than this book has room to explore.

09: handling exceptions

Before we proceed with the coverage of other features of classes in the Object Pascal language, we need to focus on one particular group of objects used to handle error conditions, known as *exceptions*.

The idea of *exception handling* is to make programs more robust by adding the capability of handling software or hardware errors (and any other type of error) in a simple and uniform way. A program can survive such errors or terminate gracefully, allowing the user to save data before exiting. Exceptions allow you to separate the error handling code from your normal code, instead of intertwining the two. You end up writing code that is more compact and less cluttered by maintenance chores unrelated to the actual programming objective.

Another benefit is that exceptions define a uniform and universal error-reporting mechanism, which is also used by the component libraries. At run time, the system raises exceptions when something goes wrong. If your code has been written properly, it can acknowledge the problem and try to solve it; otherwise, the exception is passed to its calling code, and so on. Ultimately, if no part of your code handles the exception, the system generally handles it, by displaying a standard error message

266 - 09: Handling Exceptions

and trying to continue the program. In the unusual scenario your code is executed outside of any exception handling block, raising an exception will cause the program to terminate. The whole mechanism of exception handling in Object Pascal is based on five separate keywords:

- `try` delimits the beginning of a protected block of code
- `except` delimits the end of a protected block of code and introduces the exception-handling code
- `on` marks the individual exception handling statements, tied to specific exceptions, each having the syntax `on exception-type do` statement
- `finally` is used to specify blocks of code that must always be executed, even when exceptions occur
- `raise` is the statement used to trigger an exception and has as parameter an exception object (this operation is called `throw` in other programming languages)

This is a simple comparison table of exception handling keywords in Object Pascal with languages based on the C++ exceptions syntax (like C# and Java):

<code>try</code>	<code>try</code>
<code>except on</code>	<code>catch</code>
<code>finally</code>	<code>finally</code>
<code>raise</code>	<code>throw</code>

Using the C++ language terminology, you throw an exception object and catch it by type. This is the same in Object Pascal, where you pass to the `raise` statement an exception object and you receive it as a parameter of the `except on` statements.

Try-Except Blocks

Let me start with a rather simple try-except example (part of the `ExceptionsTest` application project), one that has a general exception handling block:

```
function DividePlusOne (A, B: Integer): Integer;
begin
  try
    Result := A div B; // raises exception if B equals 0
    Inc (Result);
  except
    Result := 0;
  end;
  // more
end;
```

note When you run a program in the Delphi debugger, the debugger will stop the program by default when an exception is encountered, even if there is an exception handler. This is normally what you want, of course, because you want to know where the exception took place and you can see the call of the handler step-by-step. If you just want to let the program run when the exception is properly handled, and see what a user would see, run the program with the “Run without debugging” command or disable all (or some type of) exceptions in the debugger options.

“Silencing” the exception, as in the code above, and setting the result to zero doesn't really make a lot of sense in a real world application, but the code is meant to help you understand the core mechanism in a simple scenario. This is the code of the event handler used to call the function:

```
var
  N: Integer;
begin
  N := DividePlusOne (10, Random(3));
  Show (N.ToString);
```

As you can see the program uses a randomly generated value so that when you click the button you can be in a valid situation (2 times out of 3) or in an invalid one. This way there can be two different program flows:

- If *B* is not zero, the program does the division, executes the increment, and then skips the except block up the end statement following it (//more)
- If *B* is zero, the division raises an exception, all of the following statements are skipped (well, only one in this case) up to the first enclosing `try-except` block, which gets executed instead. After the exception block, the program won't get back to the original statement, but skips until after the except block executing the first statement after it (//more).

A way to describe this exception model is to say it follows an approach of non-resumption. In case of an error, trying to handle the error condition and getting back to the statement that caused it, is very dangerous, as the status of the program at that point is probably undefined. Exceptions significantly change the execution flow, skipping execution of the following statement and rolling back the stack until the proper error handling code is found.

The code above had a very simple `except` block, with no `on` statement. When you need to handle multiple types of exceptions (or multiple exception class types) or if you want to access the exception object passed to the block, you need to have one or more `on` statements:

```
function DividePlusOneBis (A, B: Integer): Integer;
begin
  try
    Result := A div B; // error if B equals 0
    Result := Result + 1;
```

268 - 09: Handling Exceptions

```
except
  on E: EDivByZero do
  begin
    Result := 0;
    ShowMessage (E.Message);
  end;
end;
```

In the exception-handling statement, we catch the `EDivByZero` exception, which is defined by the run-time library. There are a number of these exception types referring to run-time problems (such as a division by zero or a wrong dynamic cast), to system problems (such as out-of-memory errors), or to component errors (such as a wrong index). All of these exceptions classes inherit from the base class `Exception`, which offers some minimal features like the `Message` property I used in the code above. These classes form an actual hierarchy with some logical structure.

note Notice that while types in Object Pascal are generally marked with an initial letter `T`, exception classes take an *exception* to the rule and generally start with the letter `E`.

The Exceptions Hierarchy

Here is a partial list of the core exception classes defined in the `System.SysUtils` unit of the run-time library (most of the other system libraries add their own exception types):

```
Exception
  EArgumentException
    EArgumentOutOfRangeException
    EArgumentNullException
  EPathTooLongException
  ENotSupportedException
  EDirectoryNotFoundException
  EFileNotFoundException
  EPathNotFoundException
  EListError
  EInvalidOpException
  ENoConstructException
  EAbort
  EHeapException
    EOutOfMemory
    EInvalidPointer
  EInOutError
  EExternal
    EExternalException
  EIntError
    EDivByZero
    ERangeError
```

```

    EIntOverflow
  EMathError
    EInvalidOp
    EZeroDivide
    EOverflow
    EUnderflow
  EAccessViolation
  EPrivilege
  EControlC
  EQuit
  EInvalidCast
  EConvertError
  ECodesetConversion
  EVariantError
  EPropReadOnly
  EPropWriteOnly
  EAssertionFailed
  EAbstractError
  EIntfCastError
  EInvalidContainer
  EInvalidInsert
  EPackageError
  ECFError
  EOSError
  ESafecallException
  EMonitor
    EMonitorLockException
    ENoMonitorSupportException
  EProgrammerNotFound
  ENotImplemented
  EObjectDisposed
  EJNIException

```

note I don't know about you, but I still have to figure out the exact usage scenario of what I consider the most odd exception class, the funny `EProgrammerNotFound` exception!

Now that you have seen the core exceptions hierarchy, I can add one piece of information to the previous description of the `except-on` statements. These statements are evaluated in sequence until the system finds an exception class matching the type of the exception object that was raised. Now the matching rule used is the type compatibility rule we examined in the last chapter: an exception object is compatible with any of the base types of its own specific type (like a `TDog` object was compatible with the `TAnimal` class).

This means you can have multiple exception handler types that match the exception. If you want to be able to handle the more granular exceptions (the lower classes of the hierarchy) along with the more generic one in case none of the previous matches, you have to list the handler blocks from the more specific to the more generic (or from the child exception class up to its parent classes). Also, if you write

270 - 09: Handling Exceptions

a handler for the type `Exception` it will be a catch-all clause, so it needs to be the last of the sequence.

Here is a code snippet with two handlers in one block:

```
function DividePlusOne (A, B: Integer): Integer;
begin
  try
    Result := A div B; // error if B equals 0
    Result := Result + 1;
  except
    on EDivByZero do
      begin
        Result := 0;
        MessageDlg ('Divide by zero error',
          mtError, [mbOK], 0);
      end;
    on E: Exception do
      begin
        Result := 0;
        MessageDlg (E.Message,
          mtError, [mbOK], 0);
      end;
    end; // end of except block
  end;
end;
```

In this code there are two different exception handlers after the same `try` block. You can have any number of these handlers, which are evaluated in sequence as explained above.

Keep in mind that using a handler for every possible exception is not usually a good choice. It is better to leave unknown exceptions to the system. The default exception handler generally displays the error message of the exception class in a message box, and then resumes normal operation of the program.

tip You can actually modify the normal exception handler by providing a method for the `Application.OnException` event, for example logging the exception message in a file rather than displaying it to the user.

Raising Exceptions

Most exceptions you'll encounter in your Object Pascal programming will be generated by the system, but you can also raise exceptions in your own code when you discover invalid or inconsistent data at run time.

In most cases, for a custom exception you'll define your own exception type. Simply create a new subclass of the default exception class or one of its existing subclasses we saw above:

```
type
  EArrayFull = class (Exception);
```

In most cases, you don't need to add any methods or fields to the new exception class and the declaration of an empty derived class will suffice.

The scenario for this exception type would be a method that adds elements to an array raising an error when the array is full. This is accomplished by creating the exception object and passing it to the `raise` keyword:

```
if MyArray.IsFull then
  raise EArrayFull.Create ( 'Array full' );
```

This `Create` method (inherited from the base `Exception` class) has a string parameter to describe the exception to the user.

note You don't need to worry about destroying the object you have created for the exception, because it will be deleted automatically by the exception-handler mechanism.

There is a second scenario for using the `raise` keyword. Within an `except` block you might want to perform some actions but don't trap the exception, letting it flow to the enclosing exception handler block. In this case, you can call `raise` with no parameters. The operation is called *re-raising* an exception.

Exceptions and the Stack

When the program raises an exception and the current routine doesn't handle it, what happens to your method and function call stack? The program starts searching for a handler among the functions already on the stack. This means that the program exits from existing functions and does not execute the remaining statements. To understand how this works, you can either use the debugger or add a number of simple output lines, to be informed when a certain source code statement is executed. In the next application project, `ExceptionFlow`, I've followed this second approach.

For example, when you press the `Raise1` button in the form of the `ExceptionFlow` application project, an exception is raised and not handled, so that the final part of the code will never be executed:

```
procedure TForm1.ButtonRaise1Click(Sender: TObject);
begin
  // unguarded call
```


272 - 09: Handling Exceptions

```
AddToArray (24);  
Show ('Program never gets here');  
end;
```

Notice that this method calls the `AddToArray` procedure, which invariably raises the exception. When the exception is handled, the flow starts again after the handler and not after the code that raises the exception. Consider this modified method:

```
procedure TForm1.ButtonRaise2Click(Sender: TObject);  
begin  
  try  
    // this procedure raises an exception  
    AddToArray (24);  
    Show ('Program never gets here');  
  except  
    on EArrayFull do  
      Show ('Handle the exception');  
    end;  
    Show ('ButtonRaise1Click call completed');  
  end;  
end;
```

The last `Show` call will be executed right after the second one, while the first is always ignored. I suggest that you run the program, change its code, and experiment with it to fully understand the program flow when an exception is raised.

note Given the code location where you handle the exception is different than the one the exception was raised, it would be nice to be able to know in which method the exception was actually raised. While there are ways to get a stack trace when the exception is raised and make that information available in the handler, this is really an advanced topic I don't plan to cover here. In most cases, Object Pascal developers rely on third party libraries and tools (like JclDebug from Jedi Component Library, madExcept or EurekaLog). In addition, you have to generate and include in your code a MAP file created by the compiler and that lists the memory address of each method and function in your application.

The Finally Block

There is a fourth keyword for exception handling that I've mentioned but haven't used so far, `finally`. A `finally` block is used to perform some actions (usually cleanup operations) that should always be executed. In fact, the statements in the `finally` block are processed whether or not an exception takes place. The plain code following a `try` block, instead, is executed only if an exception was not raised or if it was raised and handled. In other words, the code in the `finally` block is always executed after the code of the `try` block, even if an exception has been raised.

Consider this method (part of the `ExceptFinally` application project), which performs some time-consuming operations and shows in the form caption its status:

```
procedure TForm1.BtnWrongClick(Sender: TObject);
var
  I, J: Integer;
begin
  Caption := 'Calculating';

  J := 0;
  // long (and wrong) computation...
  for I := 1000 downto 0 do
    J := J + J div I;

  Caption := 'Finished';
  Show ('Total: ' + J.ToString);
end;
```

Because there is an error in the algorithm (as the variable `I` can reach a value of 0 and is also used in a division), the program will break, but it won't reset the form caption. This is what a try-finally block is for:

```
procedure TForm1.BtnTryFinallyClick(Sender: TObject);
var
  I, J: Integer;
begin
  Caption := 'Calculating';
  J := 0;
  try
    // long (and wrong) computation...
    for I := 1000 downto 0 do
      J := J + J div I;
    Show ('Total: ' + J.ToString);
  finally
    Caption := 'Finished';
  end;
end;
```

When the program executes this function, it always resets the cursor, whether an exception (of any sort) occurs or not. The drawback to this version of the function is that it doesn't handle the exception.

Finally And Except

Curiously enough, in the Object Pascal language a try block can be followed by either an except or a finally statement but not both at the same time. Given you'd often want to have both blocks, the typical solution is to use two nested try blocks, associating the internal one with a finally statement and the external one with an except statement or vice versa, as the situation requires. Here is the code of this third button of the `ExceptFinally` application project:

```

procedure TForm1.BtnTryTryClick(Sender: TObject);
var
  I, J: Integer;
begin
  Caption := 'Calculating';
  J := 0;
  try try
    // long (and wrong) computation...
    for I := 1000 downto 0 do
      J := J + J div I;
    Show ('Total: ' + J.ToString);
  except
    on E: EDivByZero do
      begin
        // re-raise the exception with a new message
        raise Exception.Create ('Error in Algorithm');
      end;
    end;
  finally
    Caption := 'Finished';
  end;
end;

```

Restore the Cursor with a Finally Block

A common use case for try-finally blocks is the allocation and release of resources. Another relate case is that of a temporary configuration you need to reset after an operation has completed, even in case that operation raises an exception.

One example of a temporary configurations setting you have to restore is that of the hourglass cursor, displayed during a long operation and removed at the end restoring the original active cursor. Even if the code is simple, there is always the change an exception is raised, and therefore you should always use a try-finally block.

In the RestoreCursor application example (a VCL application, as cursor management in FireMonkey is a bit more complex) I've written the following code for saving the current cursor, temporarily setting the hourglass one, and restoring the original cursor at the end:

```

var CurrCur := Screen.Cursor;
Screen.Cursor := crHourGlass;
try
  // some slow operation
  Sleep (5000);
finally
  Screen.Cursor := CurrCur;
end;

```

Restore the Cursor with a Managed Record

To protect a resource allocation or define a temporary configuration to restore, instead of an explicit `try-finally` block you can use a manager record, which requires the compiler to add an intrinsic `finally` block. This results in writing less code to protect a resource or restore a configuration, even if there is some initial effort in defining the record.

This is a managed record representing the same behavior of the code in the previous section, saving the current cursor in a field in the `Initialize` method and resetting it in the `Finalize` method:

```
type
  THourCursor = record
  private
    FCurrCur: TCursor;
  public
    class operator Initialize (out ADest: THourCursor);
    class operator Finalize (var ADest: THourCursor);
  end;

class operator THourCursor.Initialize (out ADest: THourCursor);
begin
  ADest.FCurrCur := Screen.Cursor;
  Screen.Cursor := crHourGlass;
end;

class operator THourCursor.Finalize (var ADest: THourCursor);
begin
  Screen.Cursor := ADest.FCurrCur;
end;
```

Once you have defined this managed record

```
var HC: THourglassCursor;
    // some slow operation
    Sleep (5000);
```

note You can find more extensive examples of resource protection via managed records at the following blog post by Erik van Bilsen: <https://blog.grijjy.com/2020/08/03/automate-restorable-operations-with-custom-managed-records/>. This is part of a series of very detailed blogs on managed records.

Exceptions in the Real World

Exceptions are a great mechanism for error reporting and error handling at large (that is not within a single code fragment, but as part of a larger architecture). Exceptions in general should not be a substitute for checking a local error condition (although some developers use them this way).

For example, if you are not sure about a file name, checking if a file exists before opening is generally considered a better approach than opening the file anyway using exceptions to handle the scenario the file is not there. However, checking if there is still enough disk space before writing to the file, is a type of check that makes little sense to do all over the places, as that is an extremely rare condition.

One way to put it is that a program should check for common error conditions and leave the unusual and unexpected ones to the exception handling mechanism. Of course, the line between the two scenarios are often blurred, and different developers will have different ways to judge.

Where you'd invariably use exceptions is for letting different classes and modules pass error conditions to each other. Returning error codes is extremely tedious and error prone compared to using exceptions. Raising exceptions is more common in a component or library class than in an event handler. You can end up writing a lot of code without raising or handling exceptions.

What is extremely important and very common in every day code, instead, is using `finally` blocks to protect resources in case of an exception. You should always protect blocks that refer to external resources with a `finally` statement, to avoid resource leaks in case an exception is raised. Every time you open and close, connect and disconnect, create and destroy something within a single function or method, a `finally` statement is required. Ultimately, a `finally` statement let you keep a program stable even in case an exception is raised, letting the user continue to use or (in case of more significant issues) orderly shut down the application.

Global Exceptions Handling

If an exception raised by an event handler stops the standard flow of execution, will it also terminate the program if no exception handler is found? This is really the case for a console application or other special purpose code structures, while most visual applications (included those based on the VCL or FireMonkey libraries) have

a global message handling loop that wraps each execution in a `try-except` block, so that if an exception is raised in an event handler, this is trapped.

note Notice that if an exception is raised in the start-up code before the message loop is activated, the exceptions are generally not trapped by the library and the program will simply terminate with an error. This behavior can partially be mitigated by adding a custom `try-except` block in the main program. Still library initialization code will take place before the main program is executed and that custom `try-except` block is started.

In the general case of an exception raised during execution, what happens depends on the library, but there is a generally a programmatic way to intercept those exceptions with global handlers or a way to display an error message. While some of the details differ, this is true for both VCL and FireMonkey. In the previous demos, you saw a simple error message displayed when an exception was raised.

If you want to change that behavior you can handle the `OnException` event of the global `Application` object. Although this operation pertains more to the visual library and event handling side of the application, it is also tied to the exception handling so it is worth to cover it here.

I've taken the previous application project, called it `ErrorLog`, and I've added a new method to the main form:

```
public
procedure LogException (Sender: TObject; E: Exception);
```

In the `OnCreate` event handler I've added the code to hook a method to the global `OnException` event, and after that I've written the actual code of the global handler:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
    Application.OnException := LogException;
end;

procedure TForm1.LogException(Sender: TObject; E: Exception);
begin
    Show('Exception ' + E.Message);
end;
```

note You'll learn the details of how you can assign a method pointer to an event (like I did above) in the next chapter.

With the new method in the global exceptions handler, the program writes to the output the error message, without stopping the application with an error message.

Exceptions and Constructors

There is a slightly more advanced issue surrounding exceptions, namely what happens when an exception is raised within the constructor of an object. Not all Object Pascal programmers know that in such circumstances the destructor of that object (if available) will be called.

This is important to know, because it implies that a destructor might be called for a partially initialized object. Taking for granted that internal objects exist in a destructor because they are created in the constructor might get you into some dangerous situations in case of actual errors (that is, raising another exception before the first one is handled).

This also implies that the proper sequence for a `try-finally` should involve creating the object outside of the block, as it is automatically protected by the compiler. So if the constructor fails there is no need to `Free` the object. This is why the standard coding style in Object Pascal is to protect an object by writing:

```
AnObject := AClass.Create;
try
    // use the object...
finally
    AnObject.Free;
end;
```

note Something similar also happens for two special methods of the `TObject` class, `AfterDestruction` and `BeforeConstruction`, a pseudo-constructor and a pseudo-destructor introduced for C++ compatibility (but seldom used in Object Pascal). Notice that if the `AfterConstruction` method raises an exception, the `BeforeDestruction` method is called (and also the regular destructor).

Given I've often witnessed errors in properly disposing of an object in a destructor, let me further clarify the issue with an actual demo showing the problem... along with the actual fix. Suppose you have a class including a string list, and that you write the following code to create and destroy the class (part of the `ConstructorExcept` project):

```
type
    TObjectwithList = class
    private
        FStringList: TStringList;
    public
        constructor Create (Value: Integer);
        destructor Destroy; override;
    end;

constructor TObjectwithList.Create(Value: Integer);
```

```

begin
  if value < 0 then
    raise Exception.Create('Negative value not allowed');

    FStringList := TStringList.Create;
    FStringList.Add('one');
  end;

  destructor TObjectWithList.Destroy;
begin
  FStringList.Clear;
  FStringList.Free;
  inherited;
end;

```

At first sight, the code seems correct. The constructor is allocating the sub-object and the destructor is properly disposing it. Moreover, the calling code is written in a way that if an exception is raised after the constructor, the `Free` method is called, but if the exception is in the constructor nothing happens:

```

var
  Obj: TObjectWithList;
begin
  Obj := TObjectWithList.Create (-10);
  try
    // do something
  finally
    Show ('Freeing object');
    Obj.Free;
  end;
end;

```

So does this work? Absolutely not! When this code is involved an exception is raised in the constructor, before creating the string list, and the system immediately invokes the destructor, which tries to clear the non-existing list raising an access violation or a similar error.

Why would this happen? Again, if you reverse the sequence in the constructor (create the string list first, raise the exception later) everything works properly because the destructor indeed needs to free the string list. But that isn't a real fix, only a work around. What you should always consider is protecting the code of a destructor in a way it never assumes the constructor was completely executed. This is an example:

```

destructor TObjectWithList.Destroy;
begin
  if Assigned (FStringList) then
  begin
    FStringList.Clear;
    FreeAndNil(FStringList);
  end;
  inherited;
end;

```


Advanced Features of Exceptions

This is one of the sections of the book you might want to skip the first time you read it, unless you have already a good knowledge of the language. You can move to the next chapter and get back to this section in the future.

In the final part of the chapter, I'm going to cover some more advanced topics related with exceptions handling. I'll cover nested exceptions (`RaiseOuterException`) and intercepting exceptions of a class (`RaisingException`). These features were not part of the early versions of the Object Pascal language, and add significant power to the system.

Nested Exceptions and the InnerException Mechanism

What happens if you raise an exception within an exception handler? The traditional answer is that the new exception will replace the existing one, which is why it is a common practice to combine at least the error messages, writing code like this (lacking any actual operation, and showing only the exceptions-related statements):

```
procedure TFormExceptions.ClassicReraise;
begin
  try
    // do something...
    raise Exception.Create('Hello');
  except on E: Exception do
    // try some fix...
    raise Exception.Create('Another: ' + E.Message);
  end;
end;
```

This code is part of the `AdvancedExcept` application project. When calling the method and handling the exception, you'll see a single exception with the combined message:

```
procedure TFormExceptions.BtnTraditionalClick(
  Sender: TObject);
begin
  try
    ClassicReraise;
  except
    on E: Exception do
      Show ('Message: ' + E.Message);
  end;
end;
```

The (quite obvious) output is:

```
Message: Another: Hello
```

Now in Object Pascal there is system-wide support for nested exceptions. Within an exception handler, you can create and raise a new exception and still keep the current exception object alive, connecting it to the new exception. To accomplish this, the `Exception` class has an `InnerException` property, referring to the previous exception, and a `BaseException` property that lets you access the first exception of a series, as exception nesting can be recursive. These are the elements of the `Exception` class related to the management of nested exceptions:

```
type
  Exception = class(TObject)
  private
    FInnerException: Exception;
    FAcquireInnerException: Boolean;
  protected
    procedure SetInnerException;
  public
    function GetBaseException: Exception; virtual;
    property BaseException: Exception read GetBaseException;
    property InnerException: Exception read FInnerException;
    class procedure RaiseOuterException(E: Exception); static;
    class procedure ThrowOuterException(E: Exception); static;
  end;
```

note Static class methods are a special form of class methods. This language feature will be explained in Chapter 12.

From the perspective of a user, to raise an exception while preserving the existing one you should call the `RaiseOuterException` class method (or the identical `ThrowOuterException`, which uses C++-oriented naming). When you handle a similar exception you can use the new properties to access further information. Notice that you can call `RaiseOuterException` only within an exception handler as the *source code-based* documentation tells:

Use this function to raise an exception instance from within an exception handler and you want to "acquire" the active exception and chain it to the new exception and preserve the context. This will cause the `FInnerException` field to get set with the exception currently in play.

You should only call this procedure from within an except block where this new exception is expected to be handled elsewhere.

For an actual example you can refer to the `AdvancedExcept` application project. In this example I've added a method that raises a nested exception in the new way (compared to the `ClassicReraise` method listed earlier):

282 - 09: Handling Exceptions

```
procedure TFormExceptions.MethodWithNestedException;  
begin  
  try  
    raise Exception.Create ('Hello');  
  except  
    Exception.RaiseOuterException (  
      Exception.Create ('Another'));  
  end;  
end;
```

Now in the handler for this outer exception we can access both exception objects (and also see the effect of calling the new ToString method):

```
try  
  MethodWithNestedException;  
except  
  on E: Exception do  
    begin  
      Show ('Message: ' + E.Message);  
      Show ('ToString: ' + E.ToString);  
      if Assigned (E.BaseException) then  
        Show ('BaseException Message: ' +  
          E.BaseException.Message);  
      if Assigned (E.InnerException) then  
        Show ('InnerException Message: ' +  
          E.InnerException.Message);  
    end;  
end;
```

The output of this call is the following:

```
Message: Another  
ToString: Another  
Hello  
BaseException Message: Hello  
InnerException Message: Hello
```

There are two relevant elements to notice. The first is that in the case of a single nested exception the BaseException property and the InnerException property both refer to the same exception object, the original one. The second is that while the message of the new exception contains only the actual message, by calling ToString you get access at the combined messages of all the nested exceptions, separated by an sLineBreak (as you can see in the code of the method Exception.ToString).

The choice of using a line break in this case produces odd looking output, but once you know about it you can format it the way you like, replacing the line breaks with a symbol of your choice or assigning them to the Text property of a string list.

As a further example, let me show you what happens when raising two nested exceptions. This is the new method:

```
procedure TFormExceptions.MethodWithTwoNestedExceptions;
```

```

begin
  try
    raise Exception.Create ('Hello');
  except
    begin
      try
        Exception.RaiseOuterException (
          Exception.Create ('Another'));
      except
        Exception.RaiseOuterException (
          Exception.Create ('A third'));
      end;
    end;
  end;
end;

```

This called a method that is identical to the one we saw previously and produces the following output:

```

Message: A third
ToString: A third
Another
Hello
BaseException Message: Hello
InnerException Message: Another

```

This time the `BaseException` property and the `InnerException` property refer to different objects and the output of `ToString` spans three lines.

Intercepting an Exception

Another advanced feature added over time to the original exception handling system of the Object Pascal language is the method:

```
procedure RaisingException(P: PExceptionRecord); virtual;
```

According to the source code documentation:

This virtual function will be called right before this exception is about to be raised. In the case of an external exception, this is called soon after the object is created since the "raise" condition is already in progress.

The implementation of the function in the `Exception` class manages the inner exception (by calling the internal `SetInnerException`), which probably explains why it was introduced in the first place, at the same time as the inner exception mechanism.

In any case, now that we have this feature available we can take advantage of it. By overriding this method, in fact, we have a single post-creation function that is invariably called, regardless of the constructor used to create the exception. In other

284 - 09: Handling Exceptions

words, you can avoid defining a custom constructor for your exception class and let users call one of the many constructors of the base `Exception` class, and still have custom behavior. As an example, you can log any exception of a given class (or subclass).

This is a custom exception class (defined again in the `AdvancedExcept` application project) that overrides the `RaisingException` method:

```
type
  ECustomException = class (Exception)
  protected
    procedure RaisingException(
      P: PExceptionRecord); override;
  end;

procedure ECustomException.
  RaisingException(P: PExceptionRecord);
begin
  // log exception information
  FormExceptions.Show('Exception Addr: ' + IntToHex (
    Integer(P.ExceptionAddress), 8));
  FormExceptions.show('Exception Mess: ' + Message);

  // modify the message
  Message := Message + ' (filtered)';

  // standard processing
inherited;
end;
```

What this method implementation does is to log some information about the exception, modify the exception message and then invoke the standard processing of the base classes (needed for the nested exception mechanism to work). The method is invoked after the exception object has been created but before the exception is raised. This can be noticed because the output produced by the `Show` calls is generated before the exception is caught by the debugger! Similarly, if you put a break point in the `RaisingException` method, the debugger will stop there before catching the exception.

Again, nested exceptions and this intercepting mechanism are not commonly used in application code, as they are language features more meant for library and component developers.

10: properties and events

In the last three chapters, I have covered the foundations of OOP in Object Pascal, explaining these concepts and showing how features available in most object oriented languages are specifically implemented. Since the early days of Delphi, the Object Pascal language was a fully object-oriented language, but with a specific *flavor*. In fact, it also doubled as the language of a component-based visual development tool.

These are not disjoint features: The support for this development model is based on some core language features, like properties and events, originally introduced in Object Pascal before any other language, and later partially copied by a few OOP languages. Properties for example can be found in Java and C#, among other languages, but they do have a direct ancestry to Object Pascal... although I personally prefer the original implementation, as I'll explain shortly.

Object Pascal's ability to support rapid application development (RAD) and visual programming is the reason for concepts like properties, the published access specifier, events, the concept of a component, and a few other ideas covered in this chapter.

Defining Properties

What is a property? Properties can be described as identifiers that let you access and modify the status of an object – something that might trigger code execution behind the scenes. In Object Pascal, properties abstract and hide data access via fields or methods, making them a primary implementation of encapsulation. One way to describe properties is “*encapsulation to the max*”.

Technically, a property is an identifier with a data type that is mapped to some actual data using some `read` and `write` specifier. Differently from Java or C#, in Object Pascal the `read` and `write` specifier can be either a getter or setter method or directly a field.

For example, here is the definition of a property for a date object using a common approach (read from field, write using a method):

```
private
    FMonth: Integer;
    procedure SetMonth(Value: Integer);
public
    property Month: Integer read FMonth write SetMonth;
```

To access the value of the `Month` property, this code has to read the value of the private field `FMonth`, while, to change the value, it calls the method `SetMonth`. The code to change the value (protecting against negative values) could be something like:

```
procedure TDate.SetMonth (Value: Integer);
begin
    if Value <= 0 then
        FMonth := 1
    else
        FMonth := Value;
end;
```

note In case of incorrect input, like a negative month number it is generally better to show an error (by raising an exception) than adjust the value behind the scenes, but I'm keeping the code *as-is* for the sake of a simple introductory demo.

Notice that the data type of the field and of the property must match exactly (if there is a discrepancy, you can use a simple method for converting); similarly the type of the single parameter of a setter procedure or the return value of a getter function must exactly match the property type.

Different combinations are possible (for example, we could also use a method to read the value or directly change a field in the `write` directive), but the use of a

method to change the value of a property is most common. Here are some alternative implementations for the same property:

```
property Month: Integer read GetMonth write SetMonth;
property Month: Integer read FMonth write FMonth;
```

note When you write code that accesses a property, it is important to realize that a method might be called. The issue is that some of these methods take some time to execute; they can also produce a number of side effects, often including a (slow) repainting of the control on the screen. Although side effects of properties are seldom documented, you should be aware that they exist, particularly when you are trying to optimize your code.

The *write* directive of a property can also be omitted, making the property a *read-only* property:

```
property Month: Integer read GetMonth;
```

Technically you can also omit the *read* directive and define a *write-only* property, but that doesn't usually make much sense and is very rarely done.

Properties Compared to Other Programming Languages

If you compare this to Java or C#, in both languages the properties are mapped to methods, but the first has implicit mapping (properties are basically a convention), while the latter has explicit mapping like Object Pascal, even if only to methods:

```
// properties in Java language

private int mMonth;

public int getMonth() { return mMonth; }

public void setMonth(int value) {
    if (value <= 0)
        mMonth = 1;
    else
        mMonth = value;
}

int s = date.getMonth ();
date.setMonth (s+1);

// properties in C# language

private int mMonth;
```


288 - 10: Properties and Events

```
public int Month {  
    get { return mMonth; }  
    set {  
        if (value <= 0)  
            mMonth = 1;  
        else  
            mMonth = value;  
    }  
}  
  
date.Month++;
```

Not that I won't to discuss in depth the relative merit of properties in the various programming languages, but as I mentioned in the introduction to this chapter I think that having explicitly defined properties is a useful idea, and also that the further level of abstraction obtained by mapping properties to fields without the extra burden of a method is a very nice addition. That's why I prefer the Object Pascal implementation of properties compared to other languages.

Properties are a very sound OOP mechanism, a very well thought out application of the idea of encapsulation. Essentially, you have a name that hides the implementation of how to access the information of a class (either accessing the data directly or calling a method).

In fact, by using properties you end up with an interface that is unlikely to change. At the same time, if you only want to allow users access some fields of your class, you can easily wrap those fields into properties instead of making them public. You have no further code to write (coding simple `Get` and `Set` methods is terribly boring), and you are still able to change the implementation of your class. Even if you replace the direct data access with method-based access, you won't have to change the source code that uses these properties at all. You'll only need to recompile it. Think of this as the concept of encapsulation raised to the maximum power!

note You might wonder, if a property is defined with direct access to a private variable, doesn't that remove one of the advantages of encapsulation? The user cannot be protected against any change in the private variable's data type whereas they can be with getters and setters. However, given the user will access the data via the property, the class developer can at any time change the underlying data type and introduce getters and setters, without affecting the code using it. This is why I called this "encapsulation to the max". On the other hand, this shows the pragmatic side of Object Pascal, in that it allows the programmer a choice of any easier way (and fast code execution) where it fits the circumstances, and a smooth transition to a "proper OOP" way when that is needed.

There is one caveat in using properties in Object Pascal, though. You can usually assign a value to a property or read it, and you can freely use properties in expressions. However, you cannot pass a property as a reference parameter to a procedure

or method. This is because a property is not a memory location but an abstraction, so it cannot be used as a reference (var) parameter. As an example, unlike C# you cannot call `inc` on a property.

note A related feature, passing properties by reference, is described later in this chapter. However it is a little used feature that requires a specific compiler setting to be enabled, and is certainly not a mainstream capability.

Code Completion for Properties

If adding properties to a class might seem tedious work, the editor of the IDE lets you easily *auto-complete* properties when you write the initial part of a property declaration (within a class), like the following:

```
type
  TMyClass = class
    public
      property Month: Integer;
    end;
```

Press the `Ctrl+Shift+C` key combination while the cursor is on the property declaration and you'll get a new field added to the class along with a new setter method, with the proper mapping in the property definition and the complete implementation of the setter method, with basic code to change the field value. In other words, the code above using the keyboard shortcut (or the matching item of the local menu of the editor) becomes:

```
type
  TMyClass = class
    private
      FMonth: Integer;
    procedure SetMonth(const Value: Integer);
    public
      property Month: Integer read FMonth write SetMonth;
    end;

{ TMyClass }
procedure TMyClass.SetMonth(const Value: Integer);
begin
  FMonth := Value;
end;
```

Would you want also a getter method, replace the `read` portion of the definition with `GetMonth`, like in:

```
property Month: Integer read GetMonth write SetMonth;
```

290 - 10: Properties and Events

Now press `Ctrl+Shift+C` again and the function will also be added, but this time with no predefined code for accessing the value:

```
function TMyClass.GetMonth: Integer;  
begin  
end;
```

Adding Properties to Forms

Let's look at a specific example of encapsulation using properties. Rather than building a custom class, this time I'm going to modify the form class that the IDE generates for each visual form you create... and I'm also going to take advantage of Class Completion.

When an application has multiple forms, it is often handy to be able to access information of one form from another. You might be tempted to add a public field, but that's invariably a bad idea. Every time you want to make some information of a form available to other forms, you should rather use a property.

Simply write in the form class declaration the property name and type:

```
property Clicks: Integer;
```

Then press `Ctrl+Shift+C` to activate Code Completion. You'll see the following effect:

```
type  
  TFormProp = class(TForm)  
    private  
      FClicks: Integer;  
      procedure SetClicks(const Value: Integer);  
    public  
      property Clicks: Integer  
        read FClicks write SetClicks;  
    end;  
  implementation  
    procedure TForm1.SetClicks(const Value: Integer);  
    begin  
      FClicks := Value;  
    end;
```

Needless to say this saves you a lot of typing. Now when a user clicks on a form you can increase the click count by writing the following line, as I've done in the `OnMouseDown` event of the form of the `FormProperties` application project:

```
Clicks := Clicks + 1;
```

You might wonder, what about increasing `FClicks` directly? Well, in this specific scenario that might work, but you might as well use the `SetClicks` method to update the user interface and actually display the current value. If you bypass the property and access the field directly, the additional code in the setter method used to update user interface won't be executed and the display might get out of sync.

The other advantage of this encapsulation is that from another form you would be able to refer to the number of clicks in a properly abstracted way. In fact, properties in form classes can be used to access custom values but also to encapsulate the access the components of the form. For example, if you have a form with a label used to display some information, and you want to modify the text from a secondary form, you might be tempted to write:

```
Form1.StatusLabel.Text := 'new text';
```

This is a common practice, but it's not a good one because it doesn't provide any encapsulation of the form structure or components. If you have similar code in many places throughout an application, and you later decide to modify the user interface of the form (replacing the `StatusLabel` object with another control), you'll have to fix the code in many places.

The alternative is to use a method or, even better, a property, to hide the specific control. You can follow the steps above to add a property with both reading and writing methods, or type them in full like in:

```
property StatusText: string read GetStatusText write SetStatusText;
```

and press the Ctrl+Shift+C combination again, to let the editor add the definition of both methods for reading and writing the property:

```
function TFormProp.GetStatusText: string;
begin
    Result := LabelStatus.Text
end;

procedure TFormProp.SetStatusText(const Value: string);
begin
    LabelStatus.Text := Value;
end;
```

Notice that in this case the property is not mapped to a local field of a class, but to the field of a sub-object, the label (in case you used automatic code generation, remember to actually delete the `FStatusText` property the editor might have added on your behalf).

In the other forms of the program, you can simply refer to the `StatusText` property of the form, and if the user interface changes, only the `Set` and `Get` methods of the property are affected. Also you can even do the same inside the original form, making the code of the two properties more independent:

292 - 10: Properties and Events

```
procedure TFormProp.SetClicks(const Value: Integer);
begin
    FClicks := Value;
    StatusText := FClicks.ToString + ' clicks';
end;
```

Adding Properties to the TDate Class

In Chapter 7, I built a `TDate` class. Now we can extend it by using properties. This new application project, `DateProperties`, is basically an extension of the `viewDate` application project of Chapter 7. Here is the new declaration of the class. It has some new methods (used to set and get the values of the properties) and four properties:

```
type
    TDate = class
    private
        FDate: TDateTime;
        function GetYear: Integer;
        function GetDay: Integer;
        function GetMonth: Integer;
        procedure SetDay (const Value: Integer);
        procedure SetMonth (const Value: Integer);
        procedure SetYear (const Value: Integer);
    public
        constructor Create; overload;
        constructor Create (Y, M, D: Integer); overload;
        procedure SetValue (Y, M, D: Integer); overload;
        procedure SetValue (NewDate: TDateTime); overload;
        function LeapYear: Boolean;
        procedure Increase (NumberOfDays: Integer = 1);
        procedure Decrease (NumberOfDays: Integer = 1);
        function GetText: string; virtual;
        property Day: Integer read GetDay write SetDay;
        property Month: Integer read GetMonth write SetMonth;
        property Year: Integer read GetYear write SetYear;
        property Text: string read GetText;
    end;
```

The Year, Day, and Month properties read and write their values using corresponding methods. Here are the two related to the Month property:

```
function TDate.GetMonth: Integer;
var
    Y, M, D: Word;
begin
    DecodeDate (FDate, Y, M, D);
    Result := M;
end;
```

```

procedure TDate.SetMonth(const Value: Integer);
begin
  if (Value < 1) or (Value > 12) then
    raise EDateOutOfRange.Create ('Invalid month');
    SetValue (Year, Value, Day);
end;

```

The call to `SetValue` performs the actual encoding of the date, raising an exception in the case of an error. I've defined a custom exception class, which is raised every time a value is out of range:

```

type
  EDateOutOfRange = class (Exception);

```

The fourth property, `Text`, maps only to a read method. This function is declared as virtual, because it is replaced by the `TNewDate` subclass. There is no reason the `Get` or `Set` method of a property should not use late binding (a feature explained at length in Chapter 8).

note What is important to acknowledge in this example is that the properties do not map directly to data. They are simply computed from information stored in a different type and with a different structure than the properties seem to imply.

Having updated the class with the new properties, we can now update the example to use properties when appropriate. For example, we can use the `Text` property directly, and we can use some edit boxes to let the user read or write the values of the three main properties. This happens when the `BtnRead` button is pressed:

```

procedure TDateForm.BtnReadClick(Sender: TObject);
begin
  EditYear.Text := IntToStr (TheDay.Year);
  EditMonth.Text := IntToStr (TheDay.Month);
  EditDay.Text := IntToStr (TheDay.Day);
end;

```

The `BtnWrite` button does the reverse operation. You can write the code in either of the two following ways:

```

// direct use of properties
TheDay.Year := StrToInt (EditYear.Text);
TheDay.Month := StrToInt (EditMonth.Text);
TheDay.Day := StrToInt (EditDay.Text);

// update all values at once
TheDay.SetValue (StrToInt (EditMonth.Text),
  StrToInt (EditDay.Text),
  StrToInt (EditYear.Text));

```

The difference between the two approaches relates to what happens when the input doesn't correspond to a valid date. When we set each value separately, the program might change the year and then raise an exception and skip executing the rest of the

294 - 10: Properties and Events

code, so that the date is only partially modified. When we set all the values at once, either they are correct and are all set, or one is invalid and the date object retains the original value.

Using Array Properties

Properties generally let you access a single value, even if one of a complex data type. Object Pascal also defined array properties, or indexers as they are called in C#. An array property is a property with a further parameter of any data type that is used as an index or (more in general) a selector of the actual value.

Here is an example of definition of an array property that uses an Integer index and refers to an integer value:

```
private
  function GetValue(I: Integer): Integer;
  procedure SetValue(I: Integer; const value: Integer);
public
  property value [I: Integer]: Integer read GetValue write SetValue;
```

An array property must be mapped to read and write methods that have an extra parameter representing the index... and you can use Code Completion to define the methods as for a regular property. There are many combinations of values and indexes and a few classes in the RTL make a lot of use of array properties. For example, the `TStrings` class defines 5 of them:

```
property Names[Index: Integer]: string read GetName;
property Objects[Index: Integer]: TObject
  read GetObject write PutObject;
property Values[const Name: string]: string
  read GetValue write SetValue;
property ValueFromIndex[Index: Integer]: string
  read GetValueFromIndex write SetValueFromIndex;
property Strings[Index: Integer]: string
  read Get write Put; default;
```

While most of these array properties use the index of the string as parameter in the list, others use a string as a look-up or search value (like the `values` property above). The last of these definitions uses another important feature: it is marked with the `default` keyword. This is a powerful syntax helper: the name of an array property can be omitted, so that you can apply the square bracket operator directly to the object in question. So if you have an object `SList` of this `TStrings` type, both of the following statements will read the same value:

```
SList.Strings[1]
SList[1]
```

In other words, default array properties offer a way to define a custom `[]` operator for any object.

Setting Properties by Reference

This is a rather advanced topic (and a little used feature) you should probably skip if you are not already experienced in Object Pascal. But if you are, there are chances you never heard of this capability.

At the time the Object Pascal compiler was extended to directly support Windows COM programming, it got the ability to handle “put by ref” properties (in COM jargon) or properties that can receive a reference, rather than a value.

note “Put by ref” is the name Chris Bensen gave to this feature in his blog post introducing it: <http://chrisbensen.blogspot.com/2008/04/delphi-put-by-ref-properties.html> (Chris at the time was an R&D engineer for the product).

This is accomplished by using a *var* parameter in the setter method. Given this can lead to rather awkward scenarios, the feature (while still part of the language) is considered more of an exception than a rule, which is why it is not active by default. In other words, to enable this feature you have to ask specifically for it using a compiler directive:

```
■ {$VARPROPSETTER ON}
```

Without this directive, the following code won't compile and will issue the error “E2282 Property setters cannot take var parameters”:

```
type
  TMyIntegerClass = class
  private
    FNumber: Integer;
    function GetNumber: Integer;
    procedure SetNumber(var Value: Integer);
  public
    property Number: Integer
      read GetNumber write SetNumber;
  end;
```

This class is part of the *varProp* application project. Now what is very odd is that you can have side effects within the property setter:

```
procedure TMyIntegerClass.SetNumber(var Value: Integer);
begin
  Inc(Value); // side effect
  FNumber := Value;
end;
```


296 - 10: Properties and Events

The other very unusual effect is that you cannot assign a constant value to the property, only a variable (which should be expected, as with any call involving a parameter passed by reference):

```
var
  Mic: TMyIntegerClass;
  N: Integer;
begin
  ...
  Mic.Number := 10; // Error: E2036 Variable required
  Mic.Number := N;
```

While not a feature you'd regularly use, this is a rather advanced way of thinking about a property that lets you initialize or alter the value assigned to it. This can lead to extremely odd code like:

```
  N := 10;
  Mic.Number := N;
  Mic.Number := N;
  Show(Mic.Number.ToString);
```

The two consecutive, identical assignments look rather odd, but they do produce a side effect, turning the actual number into 12. Probably the most convoluted and nonsensical way to obtain that result!

The published Access Specifier

Along with the `public`, `protected`, and `private` access directives (and the less-commonly used `strict private` and `strict protected`), the Object Pascal language has another very peculiar one, called `published`. A `published` property (or field or method) is not only available at run time like a `public` one, but it generates extended run-time information (RTTI) that can be queried.

In a compiled language, in fact, the compiled symbols are processed by the compiler and can be used by the debugger while testing the application, but generally leave no track at run-time. In other words (at least in the early times of Object Pascal) if a class has a property called `Name` you can use it in your code to interact with the class, but you have no way to figure out if a class has a property matching a given string, "`Name`".

note Both the Java and the C# languages are compiled languages that take advantage of a complex virtual execution environment and for this reason they have extensive run-time information, generally called *reflection*. The Object Pascal language introduced reflection (also called extended RTTI) after some years, so it still has both some basic RTTI tied to the published keyword explored in this chapter and a more comprehensive form of reflection covered in Chapter 16.

Why is this extra information about a class required? It is one of the foundations of the component model and visual programming model the Object Pascal libraries rely upon. Some of this information is used at design time in the development environment, to populate the Object Inspector with a list of properties a component provides. This isn't a hard coded list, but something generated via run-time inspection of the compiled code.

Another example, probably a little too complex to fully delve into now, is the streaming mechanism behind the creation of the FMX and DFM files and any accompanying visual forms. Streaming will be introduced only in Chapter 18 because it is more part of the run-time library than of the core language.

To summarize the concept, a regular use of the `published` keyword is important when you write components to be used by yourself or others in the development environment. Usually, the `published` portions of a component contains only properties, while form classes also use published fields and methods, as I'll cover later.

Design-Time Properties

We have seen earlier in this chapter that properties play an important role for the encapsulation of the data of a class. They also play a fundamental role in enabling the visual development model. In fact, you can write a component class, make it available in the development environment, create an object by adding it to a form or a similar design surface, and interact with its properties with the Object Inspector. Not all properties can be used in this scenario, only those marked as published in the component class. This is why Object Pascal programmers make a distinction between *design-time* properties and *run-time only* properties.

Design-time properties are those declared in a `published` section of the class declaration, and can be used at design time in the IDE, and also in code. Any other property that is declared in the `public` section of the class is not available at design time but only in code, and it is often called *run-time only*.

In other words, you can see the value and change the values of `published` properties at design time using the Object Inspector. This is the tool that the visual programming environment provides to let you access properties. At run time, you can access

298 - 10: Properties and Events

any `public` or `published` property of another class in the exact same way by reading or writing its value in code.

Not all of the classes have properties. Properties are present in components and in other subclasses of the `TPersistent` class, because properties usually can be streamed and saved to a file. A form file, in fact, is nothing but a collection of published properties of the components on the form.

To be more precise, you don't need to inherit from `TPersistent` to support the concept of a published section, but you need to compile a class with the `$M` compiler directive. Each class compiled with that directive, or derived from a class compiled with it, supports the published section. Given `TPersistent` is compiled with this setting, any derived class has this support.

note The following two sections on default visibility and automatic RTTI add further information to the effect of the `$M` directive and the `published` keyword.

Published and Forms

When the IDE generates a form, it places the definitions of its components and methods in the initial portion of its definition, before the `public` and `private` keywords. These fields and methods of the initial portion of the class are `published`. The default is `published` when no special keyword is added before an element of a component class.

note To be more precise, `published` is the default keyword only if the class was compiled with the `$M+` compiler directive or is descended from a class compiled with `$M+`. As this directive is used in the `TPersistent` class, most classes of the library and all of the component classes default to `published`. However, non-component classes (such as `TStream` and `TList`) are compiled with `$M-` and default to public visibility.

Here is an example:

```
type
  TForm1 = class(TForm)
    Memo1: TMemo;
    BtnTest: TButton;
```

The methods assigned to any event should be `published` methods, and the fields corresponding to your components in the form should be `published` to be automatically connected with the objects described in the form file and created along with the form. Only the components and methods in the initial `published` part of your form declaration can show up in the Object Inspector (in the list of components of

the form or in the list of the available methods displayed when you select the drop-down list for an event).

Why should the components of a class be declared with published field, while they could be private and better follow OOP encapsulation rules? The reason lies in the fact that these components are created by reading their streamed representation, but once they are created they need to be assigned to the corresponding form fields.

This is done using the RTTI generated for published fields (which was originally the only kind of RTTI available in Object Pascal until extended RTTI, covered in Chapter 16, was introduced).

note Technically it is not really compulsory to use published fields for components. You can make your code more OOP savvy by making them private. However, this does require extra run-time code. I'll explain this a little more in the last section of this chapter, "RAD and OOP".

Automatic RTTI

Another special behavior of the Object Pascal compiler, is that if you add the published keyword to a class, which doesn't inherit from `TPersistent`, the compiler will automatically enable RTTI generation automatically adding the `{$M+}` behavior.

Suppose you have this class:

```
type
  TMyTestClass = class
    private
      FValue: Integer;
      procedure SetValue(const Value: Integer);
    published
      property Value: Integer read FValue write SetValue;
    end;
```

The compiler shows a warning like:

```
[dcc32 warning] AutoRTTIForm.pas(27): W1055 PUBLISHED caused RTTI ($M+)
to be added to type 'TMyTestClass'
```

What happens is that the compiler automatically injects the `{$M+}` directive in the code, as you can see in the `AUTO RTTI` application project, which include the code above. In this program, you can write the following code, which accesses a property dynamically (using the old-fashioned `System.TTypeInfo` unit):

```
uses
  TypInfo;

procedure TFormAutoRtti.BtnTetClick(Sender: TObject);
var
```

300 - 10: Properties and Events

```
Test1: TMyTestClass;  
begin  
  Test1 := TMyTestClass.Create;  
  try  
    Test1.Value := 22;  
    Memo1.Lines.Add (GetPropValue (Test1, 'Value'));  
  finally  
    Test1.Free;  
  end;  
end;
```

note Although I'll occasionally use the `TypeInfo` unit and functions like `GetPropValue` defined in it, the real power of RTTI access is given by the more modern `RTTI` unit and its extensive support for reflection. Given this is a rather complex topic, I felt it important to devote a separate chapter to it, and also to distinguish the two *flavors* of RTTI that Object Pascal supports.

Event-Driven Programming

In a component-based library (but also in many other scenarios) the code you write is not just a flat sequence of actions, but mostly a collection of reactions. By that I mean you define how the application should “react” when something happens. This “something” could be a user operation, such as clicking on a button, a system operation, a change in the status of a sensor, some data becoming available over a remote connection, or just about anything else.

These external or internal triggers of actions are generally called events. Events were originally a mapping of message-oriented operating systems, like Windows, but have come a long way since that original concept. In modern libraries, in fact, most events are triggered internally when you set properties, call methods, or interact with a given component (or indirectly with another one).

How do events and event-driven programming relate with OOP? The two approaches are different in respect to when and how you create a new inherited class compared to using a more general one.

In a pure form of object-oriented programming, whenever an object has a different behavior (or a different method) than another one, it should belong to a different class. We have seen this in a few demos.

Now let's consider this scenario. A form has four buttons. Each button requires a different behavior when you click on it. So in pure OOP terms, you should have four different button subclasses each with a different version of a “*click*” method. This

approach is formally correct, but it would require a lot of extra code to write and maintain, increasing the complexity.

Event-driven programming considers a similar scenario and suggests the developer to add some behavior to button objects that are all of the same class. The behavior becomes a decoration or extension of the object status, without requiring a new class. The model is also called delegation, because the behavior of an object is delegated to a method of a class other than the object's own class.

Events are implemented in different ways by different programming languages, for example:

- Using references to methods (called method pointers as in Object Pascal) or to event objects with an internal method (as happens in C#)
- Delegating events code to a specialized class implementing an interface (like it generally happens in Java)
- Using closures like it generally happens in JavaScript (an approach Object Pascal also supports with anonymous methods, covered in Chapter 15), although in JavaScript all methods are closures so the differences between the two concepts are a bit blurred in that language.

The concept of events and event-driven programming has become quite common and it is supported by many different programming languages and user-interface libraries. However, the way Delphi implements support for events is quite unique. The following section explains the technology behind it in details.

Method Pointers

We have seen in the last part of Chapter 4 that the language has the concept of function pointers. This is a variable holding the memory location of a function, that you can use to call the function indirectly. A function pointer is declared with a specific signature (as set of parameter types and a return type, if any).

Similarly, the language has the concept of method pointers. A method pointer is a reference to the memory location of a method that belongs to a class. Like a function pointer type, the method pointer type has a specific signature. However, a method pointer carries over some more information, that is the object the method will be applied to (or the object that will be used as `self` parameter when the method is called).

In other words, a method pointer is a reference to a method (at a specific memory address) for one specific object in memory. When you assign a value to a method

302 - 10: Properties and Events

pointer, you have to refer to a method of a given object, i.e. a method of a specific instance!

note You can better understand the implementation of a method pointer if you look at the definition of the data structure often used at the low-level to express this construct, which is called `TMethod`. This record has two fields `Code` and `Data`, representing respectively the method address and the object it will be applied to. In other similar languages the code reference is captured by a delegate class (C#) or a method of an interface (Java).

The declaration of a method pointer type is similar to that of a procedural type, except that it has the keywords of `object` at the end of the declaration:

```
type
  IntProceduralType = procedure (Num: Integer);
  TStringEventType = procedure (const S: string) of object;
```

When you have declared a method pointer, such as the one above, you can declare a variable of this type and assign to it a compatible method of any object. What's a compatible method? One that has the same parameters as those requested by the method pointer type, such as a single string parameter in the example above.

note The reference to a method of any object can be assigned to a method pointer as long as it is compatible with the method pointer type.

Now that you have a method pointer type, you can declare a variable of this type and assign a compatible method to it:

```
type
  TEventTest = class
    public
      procedure ShowValue (const S: string);
      procedure UseMethod;
    end;

procedure TEventTest.ShowValue (const S: string);
begin
  Show (S);
end;

procedure TEventTest.UseMethod;
var
  StringEvent: TStringEventType;
begin
  StringEvent := ShowValue;
  StringEvent ('Hello');
end;
```

Now this simple code doesn't really explain the usefulness of events, as it focused on the low-level method pointer type concept. Events are based on this technical implementation, but go beyond it by storing a method pointer in one object (say, a

button) to refer to a method of a different object (say, the form with the `OnClick` handler for the button). In most cases, events are also implemented using properties.

note While it is much less common, in Object Pascal you can also use anonymous methods to define an event handler. The reason this is less common is probably as the feature was introduced fairly recently in the language, and many libraries already existed at that point. Moreover, it adds a little extra complexity. You can find an example of that approach in Chapter 15. Another possible extension is the definition of multiple handlers for a single event, like C# supports, which is not a standard feature but one that you could implement yourself.

The Concept of Delegation

At first glance, the goal of this technique may not be clear, but this is one of the cornerstones of Object Pascal component technology. The secret is in the word *delegation*. If someone has built an object that has some method pointers, you are free to change the object's behavior simply by assigning new methods to the pointers. Does this sound familiar? It should.

When you add an `OnClick` event handler for a button, the development environment does exactly that. The button has a method pointer, named `OnClick`, and you can directly or indirectly assign a method of the form to it. When a user clicks the button, this method is executed, even if you have defined it inside another class (typically, in the form).

What follows is a listing that outlines the code actually used in Delphi libraries to define the event handler of a button component and the related method of a form:

```

type
  TNotifyEvent = procedure (Sender: TObject) of object;

  TMyButton = class
    OnClick: TNotifyEvent;
  end;

  TForm1 = class (TForm)
    procedure Button1Click (Sender: TObject);
    Button1: TMyButton;
  end;

var
  Form1: TForm1;

```

Now inside a procedure, you can write

```

MyButton.OnClick := Form1.Button1Click;

```


304 - 10: Properties and Events

The only real difference between this code fragment and the code of the library is that `OnClick` is a property name, and the actual data it refers to is called `FOnClick`. An event that shows up in the Events page of the Object Inspector, in fact, is nothing more than a property that is a method pointer. This means, for example, that you can dynamically modify the event handler attached to a component at design time or even build a new component at run time and assign an event handler to it.

The `DynamicEvents` application project showcases both scenarios. The form has a button with a standard `OnClick` event handler associated with it. However I've added a second public method to the form with the same signature (the same parameters):

```
public  
procedure BtnTest2Click(Sender: TObject);
```

When the button is pressed, beside displaying a message it switches the event handler to the second one, changing the future behavior of the click operation:

```
procedure TForm1.BtnTestClick(Sender: TObject);  
begin  
    ShowMessage ('Test message');  
    BtnTest.OnClick := BtnTest2Click;  
end;  
  
procedure TForm1.BtnTest2Click(Sender: TObject);  
begin  
    ShowMessage ('Test message, again');  
end;
```

Now the first time you press the button, the first (default) event handler is executed, while any other time you get the second event handler to run.

note As you type the code to assign a method to an event, code completion will suggest the available event name to you, and turn it into an actual function call with parentheses at the end. This is not correct. You have to assign to the event the method itself, without calling it. Other wise the compiler will try to assign the result of the method call (which being a procedure, doesn't exist) resulting in an error.

The second part of the project demonstrates a completely dynamic event association. As you click on the surface of the form, a new button is created dynamically with an event handler that shows the caption of the associated button (the `Sender` object):

```
procedure TForm1.BtnNewClick(Sender: TObject);  
begin  
    ShowMessage ('You selected ' + (Sender as TButton).Text);  
end;  
  
procedure TForm1.FormMouseDown(Sender: TObject; Button: TMouseButton;  
    Shift: TShiftState; X, Y: Single);
```

```

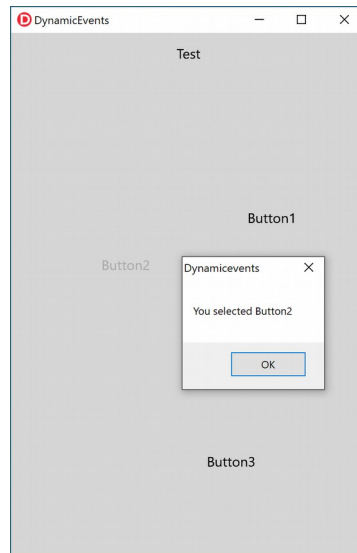
var
  AButton: TButton;
begin
  AButton := TButton.Create(Self);
  AButton.Parent := Self;
  AButton.SetBounds(X, Y, 100, 40);
  Inc (FCounter);
  AButton.Text := 'Button' + IntToStr (FCounter);
  AButton.OnClick := BtnNewClick;
end;

```

With this code each of the dynamically created buttons will react to a click of the mouse by showing a message that depends on the button, even if using a single event handler, thanks to the use of the Sender parameter of the event. An example of the output is visible in Figure 10.1.

Figure 10.1:

The message displayed by dynamically created buttons in the DynamicEvents application project



Events Are Properties

A very important concept is that events in Object Pascal are almost invariably implemented as method pointer type properties. This means that to handle an event of a component, you assign a method to the corresponding event property. In terms of code, this means you can assign to an event handler a method of an object, using code like the following that we have already seen in the previous section:

306 - 10: Properties and Events

```
| Button1.OnClick := ButtonClickHandler;
```

Again, the rule is that the method pointer type of the event must match the signature of the method you assign, or the compiler will issue an error. The system defined several standard method pointer types for events, that are commonly used, starting with the simple:

```
| type  
| TNotifyEvent = procedure (Sender: TObject) of object;
```

This is generally the type of the `OnClick` event handler, so this implies the method above must be declared (within a class) as:

```
| procedure ButtonClickHandler (Sender: TObject);
```

If this sounds a bit confusing, consider what happens in the development environment. You select a button, let's say `Button1`, double-click on it and on the `OnClick` event listed in the Object Inspector of the development environment, and a new empty method is added to the container module (likely a form):

```
| procedure TForm1.Button1Click (Sender: TObject)  
| begin  
|  
| end;
```

You fill in the method's code, and voila, everything works. This is because the assignment of the event handler method to the event happens behind the scenes exactly in the same way all other properties you set at design time are applied to the components.

From the description above you can understand there is no one-to-one correspondence between an event and the method assigned to it. Quite the opposite. You can have several events that share the same event handler, which explains the reason for the frequently used `Sender` parameter, which indicates which of the objects triggered the event. As an example, if you have the same `OnClick` event handler for two buttons, the `Sender` value will contain a reference to the button object that was clicked.

note You can assign the same method to different events in code, as shown above, but also at design time. When you select an event in the Object Inspector, you can press the arrow button on the right of the event name to see a drop-down list of “compatible” methods—a list of methods having the same method pointer type. This makes it is easy to select the same method for the same event of different components. In some cases, you can also assign the same handler to different compatible events of the same component.

Adding an Event to the TDate Class

As we've added some properties to the `TDate` class, we can now add an event. The event is going to be very simple. It will be called `OnChange`, and it can be used to warn the user of the component that the value of the date has changed. To define an event, we simply define a property corresponding to it, and we add some data to store the actual method pointer the event refers to. These are the new definitions added to the class in the `DateEvent` application project:

```
type
  TDate = class
  private
    FOnChange: TNotifyEvent;
    ...
  protected
    procedure DoChange; dynamic;
    ...
  public
    property OnChange: TNotifyEvent
      read FOnChange write FOnChange;
    ...
end;
```

The property definition is actually very simple. A developer using this class can assign a new value to the property and, hence, to the `FOnChange` private field. The field is generally unassigned when the program starts: event handlers are for users of the component, not the component writer. A component writer needing some behavior will add it to the component methods.

In other words, the `TDate` class simply accepts an event handler and calls the method stored in the `FOnChange` field when the value of the date changes. Of course, the call takes place only if the event property has been assigned.

The `DoChange` method (declared as a `dynamic` method as is traditional with event firing methods) makes the test and the method call:

```
procedure TDate.DoChange;
begin
  if Assigned (FOnChange) then
    FOnChange (Self);
end;
```

note As you might remember from Chapter 8, a dynamic method is similar to a virtual method, but uses a different implementation that reduced the memory footprint to the expense of a slightly slower call.

The `DoChange` method in turn is called every time one of the values changes, as in the following code:

308 - 10: Properties and Events

```
procedure TDate.SetValue (Y, M, D: Integer);  
begin  
    FDate := EncodeDate (Y, M, D);  
    // Fire the event  
    DoChange;  
end;
```

Now if we look at the program that uses this class, we can simplify its code considerably. First, we add a new custom method to the form class:

```
type  
    TDateForm = class(TForm)  
    ...  
    procedure DateChange(Sender: TObject);
```

The code of this method simply updates the label with the current value of the `Text` property of the `TDate` object:

```
procedure TDateForm.DateChange;  
begin  
    LabelDate.Text := TheDay.Text;  
end;
```

This event handler is then installed in the `FormCreate` method:

```
procedure TDateForm.FormCreate(Sender: TObject);  
begin  
    TheDay := TDate.Init (7, 4, 1995);  
    LabelDate.Text := TheDay.Text;  
    // Assign the event handler for future changes  
    TheDay.OnChange := DateChange;  
end;
```

Well, this seems like a lot of work. Was I lying when I told you that the event handler would save us some coding? No. Now, after we've added some code, we can completely forget about updating the label when we change some of the data of the object. Here, as an example, is the handler of the `OnClick` event of one of the buttons:

```
procedure TDateForm.BtnIncreaseClick(Sender: TObject);  
begin  
    TheDay.Increase;  
end;
```

The same simplified code is present in many other event handlers. Once we have installed the event handler, we don't have to remember to update the label continually. That eliminates a significant potential source of errors in the program. Also note that we had to write some code at the beginning because this is not a component installed in the development environment but simply a class. With a component, you simply select the event handler in the Object Inspector and write a single line of code to update the label. That's all.

This brings us to the question, how difficult is it to write a new component in Delphi? It's actually so simple that I'm going to show you how to do it in the next section.

note This is meant to be just a short introduction to the role of properties and events and to writing components. A basic understanding of these features is important for every Delphi developer. This book doesn't delve into the detail of writing custom components.

Creating a TDate Component

Now that we understand properties and events, the next step is to see what is a component. We'll briefly explore this topic by turning our `TDate` class into a component. First, we have to inherit our class from the `TComponent` class, instead of the default `TObject` class. Here is the code:

```
type
  TDate = class (TComponent)
  ..
  public
    constructor Create (AOwner: TComponent); overload; override;
    constructor Create (Y, M, D: Integer); reintroduce; overload;
```

As you can see, the second step was to add a new constructor to the class, overriding the default constructor for components to provide a suitable initial value. Because there is an overloaded version, we also need to use the `reintroduce` directive for it, to avoid a warning message from the compiler. The code of the new constructor simply sets the date to today's date, after calling the base class constructor:

```
constructor TDate.Create (AOwner: TComponent);
var
  Y, D, M: Word;
begin
  inherited Create (AOwner);
  // today...
  FDate := Date;
```

Having done this, we need to add to the unit that defines our class (the `Dates` unit of the `DateComp` application project) a `Register` procedure. (Make sure this identifier start with an uppercase `R`, otherwise it won't be recognized.) This is required in order to add the component to the IDE. Simply declare the procedure, which requires no parameters, in the `interface` portion of the unit, and then write this code in the `implementation` section:

```
procedure Register;
```

310 - 10: Properties and Events

```
begin  
  RegisterComponents ( 'Sample', [TDate]);  
end;
```

This code adds the new component to the *Sample* page of the Tools Palette, creating the page if necessary.

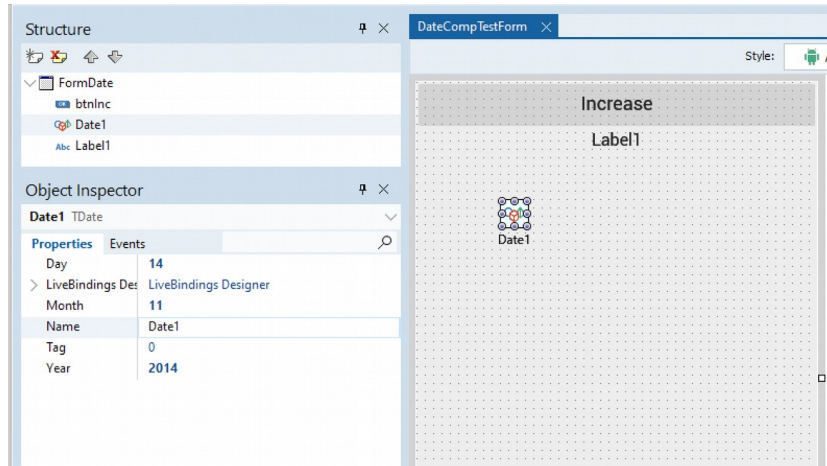
The last step is to install the component. For this we need to create a package, which is a special type of application project hosting components. All you have to do is:

- Select the File | New | Other menu of the IDE, opening the New Items dialog box
- Select “Package”
- Save the package with a name (possibly in the same folder with the unit with the actual component code)
- In the newly created package project, in the Project Manager pane, right click on the Contains node to add a new unit to the project and select the unit with the TDate component class.
- Now in the Project Manager you can right click on the package node and not only issue a Build command but also select the Install menu item, to install the component in the development environment
- If you start with the code that comes with the book, all you have to do is the last step of the sequence above: Open the DatePackage project from the DateComponent folder and compile and install it.

If you now create a new project and move to the Tools Palette, you should see the new component under *Sample*. Simply start typing its name to search for it. This will be shown using the default icon for components. At this point you can place the component on a form and start manipulating its properties in the Object Inspector, as you can see in Figure 10.2. You can also handle the onChange event in a much easier way than in the last example.

Figure 10.2:

The properties of our new TDate component in the Object Inspector



Besides trying to build your own sample application using this component (something I really suggest you do), you can now open the DateComponent application project, which is an updated version of the component we've built step-by-step over the last few sections of this chapter. This is basically a simplified version of the DateEvent application project, because now the event handler is directly available in the Object Inspector.

note If you open the DateCompTest application project before compiling and installing the specific component package (DatePackage application project), the IDE won't recognize the component as it opens the form and will give you an error message. You won't be able to compile the program or make it work properly until you install the new component.

Implementing Enumeration Support in a Class

In Chapter 3 we have seen how you can use a `for-in` loop as an alternative to a classic `for` loop. In that section I described how you can use `for-in` loops for arrays, strings, sets and some other system data types. It is possible to apply such a loop to any class, as long as it defines enumeration support. While the most obvious example will be classes that contain lists of elements, from a technical point of view this feature is quite open ended.

312 - 10: Properties and Events

There are two rules you have to follow to support enumerating the elements of a class in Object Pascal: add a method call `GetEnumerator` that returns a class (the actual enumerating class); define this enumerating class with a `Next` method and a `Current` property, the first for navigating among elements and the second for returning an actual element. Once this is done (and I'll show you how in an actual example in a second) the compiler can resolve a `for-in` loop, in which the target is our class and the individual elements must be of the same type of the `Current` property of the enumerator.

Although it is not strictly necessary, it seems a good idea to implement the enumerator support class as a nested type (a language feature covered in Chapter 7) because it really makes no sense to use the specific type used for the enumeration by itself.

The following class, part of the `NumbersEnumerator` application project, stores a range of numbers (a kind of abstract collection) and allows iterating over them. This is made possible by defining an enumerator, declared as nested type and returned by the `GetEnumerator` function:

```
type
  TNumbersRange = class
  public
    type
      TNumbersRangeEnum = class
      private
        NPos: Integer;
        FRange: TNumbersRange;
      public
        constructor Create (aRange: TNumbersRange);
        function MoveNext: Boolean;
        function GetCurrent: Integer;
        property Current: Integer read GetCurrent;
      end;

  private
    FNStart: Integer;
    FNEnd: Integer;
  public
    function GetEnumerator: TNumbersRangeEnum;
    procedure Set_NEnd(const Value: Integer);
    procedure Set_NStart(const Value: Integer);

    property NStart: Integer read FNStart write Set_NStart;
    property NEnd: Integer read FNEnd write Set_NEnd;
  end;
```

The `GetEnumerator` method creates an object of the nested type that stores status information for iterating over the data.

Notice how the enumerator constructor keeps a reference to the actual object it is enumerating (object that is passed as parameter using `self`) and sets the initial position to the very beginning:

```
function TNumbersRange.GetEnumerator: TNumbersRangeEnum;
begin
    Result := TNumbersRangeEnum.Create (self);
end;

constructor TNumbersRange.TNumbersRangeEnum.
    Create(ARange: TNumbersRange);
begin
    inherited Create;
    FRange := ARange;
    NPos := FRange.NStart - 1;
end;
```

note Why does the constructor set the initial value to the first value minus 1, rather than the first value, as expected? It turns out the compiler generated code for the `for in` loop corresponds to creating the enumeration and executing the code *while Next do use Current*. The test is performed before getting the first value, as the list may have no value. This implies that `Next` is called before the first element is used. Rather than implementing this with more complex logic, I've simply set the initial value to *one before the first* so the first time `Next` is called the enumerator is positioned on the first value.

Finally, the enumerator methods let you access the data and provide a way to move to the next value in the list (or the next element within the range):

```
function TNumbersRange.TNumbersRangeEnum.
    GetCurrent: Integer;
begin
    Result := NPos;
end;

function TNumbersRange.TNumbersRangeEnum.
    MoveNext: Boolean;
begin
    Inc (NPos);
    Result := NPos <= FRange.NEnd;
end;
```

As you can see in the code above the `Next` method serves two different purposes, moving to the following element of the list and checking if the enumerator has reached the end, in which case the method returns false.

After all this work, you can now use the `for..in` loop to iterate through the values of the range object:

```
var
    ARange: TNumbersRange;
    I: Integer;
begin
```

314 - 10: Properties and Events

```
ARange := TNumbersRange.Create;  
ARange.nStart := 10;  
ARange.nEnd := 23;  
  
for I in ARange do  
  Show (IntToStr (I));
```

The output is simply the list of values *enumerated* between 10 and 23 inclusive:

```
10  
11  
12  
13  
14  
15  
16  
17  
18  
19  
20  
21  
22  
23
```

note There are many cases in which the RTL and VCL libraries define enumerators, for example each TComponent can enumerate the component it owns. What is lacking is child controls enumeration. In Chapter 12, in the section “Adding an Enumeration with a Class Helper”, we’ll see how you can create one. The reason the example is not here is we first need to discuss class helpers.

15 Tips About Mixing RAD and OOP

In this chapter I covered properties, events, and the published keyword, that make up the core language features related to rapid application development (RAD) or visual development or event-driven programming (three terms that refer to the same conceptual model). While this is a very powerful model, it is backed by a solid OOP architecture. At times, the RAD approach could push developers to forget about good OOP practices. At the same time, going back to writing pure code, ignoring the RAD approach can often be counterproductive. In this last section of the chapter I’ve listed a few tips and suggestion about bridging the two approaches. Another way to describe it is to consider it a section on “the OOP beyond RAD”.

note The material in this final section of the chapter was originally published in the issue 17 of “The Delphi Magazine” (July 1999) with the title “20 Rules for OOP in Delphi”. Now I trimmed a few of the rules and reworded others, but the essence remains.

Tip 1: A Form is a Class

Programmers often treat forms as objects, while in fact they are classes. The difference is that you can have multiple form objects based on the same form class.

The confusing thing is that the IDE creates a default global variable and (depending on your settings) can also create a form object at startup for every form class you define in your project. This is certainly handy for newcomers, but is generally a bad habit for any non-trivial application.

Of course, it is very important to give a meaningful name to each form (and its class) and each unit. Unluckily the two names must be different, but you can use a convention to map the two in a consistent way (such as `AboutForm` and `About.pas`).

As you'll progress through the following steps you'll see the practical effects of this “a form is a class” concept.

Tip 2: Name Components

It's important to use descriptive names for components, too. The most common notation is to use a few lower case initial letters for the class type, followed by the role of the component, as in `BtnAdd` or `EditName`. There are actually many similar notations following this style and there is really no reason to say any one of them is best, it's up to your personal taste.

Tip 3: Name Events

It is equally important to give proper names to event handling methods. If you name the components properly, the default name of `Button1Click`, for example, becomes `BtnAddClick`. Although we can guess what the method does from the button name, I think it is way better to use a name describing the effect of the method, not when the method is triggered. For example, the `OnClick` event of the `BtnAdd` button could be named `AddToList` if this is what the method does.

316 - 10: Properties and Events

This makes the code more readable, particularly when you call the event handler from another method of the class, and helps developers attach the same method to multiple events or to different components, although I have to say that using `Actions` is the preferred choice for attaching a single event to multiple user interface elements in any non-trivial programs.

note `Actions` and the `ActionList` component are a very nice architectural features of both VCL and FireMonkey UI libraries, offering a conceptual separation of user operations (and their status) from the user interface controls they are associated with. Activating the control, executes the action. But if you logically disable the action, the UI elements also gets disabled. This topic is beyond the scope of this book, but worth investigating if you use any of those frameworks.

Tip 4: Use Form Methods

If forms are classes their code is collected in methods. Besides the event handlers, which play a special role but can still be called as other methods, it is often useful to add custom methods to form classes. You can add methods performing actions and accessing to the status of the form. It is much better to add a public method to a form than to let other forms operate on its components directly.

Tip 5: Add Form Constructors

A secondary form created at runtime can provide other specific constructors beside the default one (inherited from the `TComponent` class). My suggestion is to overload the `Create` method, adding the required initialization parameters, as in the following code snippet:

```
public
  constructor Create (Text: string); reintroduce; overload;

constructor TFormDialog.Create(Text: string);
begin
  inherited Create (Application);
  Edit1.Text := Text;
end;
```

Tip 6: Avoid Global Variables

Global variables (that is, variables declared in the interface portion of a unit) should be avoided. Here are a few suggestions to help you do this. If you need extra data

storage for a form, add some private fields to it. In this case each form instance will have its own copy of the data.

You might use unit variables (declared in the implementation portion of the unit) for data shared among multiple instances of the form class, but it is much better to use class data (explained in Chapter 12).

Tip 7: Never Use Form1 In TForm1 Methods

You should never refer to a specific object in a method of the class of that object. In other words, never refer to `Form1` in a method of the `TForm1` class. If you need to refer to the current object, use the `self` keyword. Keep in mind that most of the time this is not needed, as you can refer directly to methods and data of the current object. If you don't follow this rule, you'll get into real trouble when you create multiple instances of the form.

Tip 8: Seldom Use Form1 In Other Forms

Even in the code of other forms, try to avoid direct references to global objects, such as `Form1`. It is much better to declare local variables or private fields to refer to other forms. For example, the main form of a program can have a private field referring to a dialog box. Obviously this rule becomes essential if you plan creating multiple instances of the secondary form. You can keep a list in a dynamic array of the main form, or simply use the `Forms` array of the global `Screen` object to refer to any form currently existing in the application.

Tip 9: Remove the Global Form1 Variable

Actually, my suggestion is to remove the global form object which is automatically added by the IDE to the project as you add a new form to it, like `Form1`. This is possible only if you disable the automatic creation of that form, something which I suggest you should get rid of anyway. Alternatively, you can delete the corresponding line used to create the form from the project source code.

Needless to say if the form is not created automatically, you'll need some code in your application to create it, and possibly some other variable to refer to it.

I think that removing the global form object is very useful for Object Pascal newcomers, who then won't get confused between the class and the global object

318 - 10: Properties and Events

anymore. In fact, after the global object has been removed, any reference to it will result in an error.

Tip 10: Add Form Properties

As I've already discussed in the section "Adding Properties to Forms" in this chapter, when you need data for a form, add a private field. If you need to access this data from other classes, then add properties to the form. With this approach you will be able to change the code of the form and its data (including its user interface) without having to change the code of other forms or classes. You should also use properties or methods to initialize a secondary form or dialog box, and to read its final state. The initialization can also be performed using a constructor, as I have already described.

Tip 11: Expose Components Properties

When you need to access the status of another form, you should not refer directly to its components. This would bind the code of other forms or classes to the user interface, which is one of the portions of an application subject to most changes. Rather, declare a form property mapped to the component property: this is accomplished with a `Get` method that reads the component status and a `Set` method that writes it. Suppose you now change the user interface, replacing the component with another one. All you have to do is fix the `Get` and `Set` methods related with the property, you won't have to check and modify the source code of all the forms and classes which might refer to that component.

Tip 12: Use Array Properties When Needed

If you need to handle a series of values in a form, you can declare an array property. In case this is an important information for the form you can make it also the default array property of the form, so that you can directly access its value by writing `SpecialForm[3]`. This was already covered for a more generic case than forms in the section "Using Indexed Properties".

Tip 13: Starting Operations In Properties

Remember that one of the advantages of using properties instead of accessing global data is that you can call methods and do any operation when writing (or reading) the value of a property. For example, you can draw directly on the form surface, set the values of multiple properties, call special methods, change the status of multiple components at once, or fire an event, if available.

Another related example is to use property getters to implement delayed creation. Rather than creating a sub-object in the class constructor, you can create it the first time it is needed, writing code like:

```
private
  FBitmap: TBitmap;
public
  property Bitmap: TBitmap read GetBitmap;

function TBitmap.GetBitmap: TBitmap;
begin
  if not Assigned (FBitmap) then
    FBitmap := ... // create it and initialize it
  Result := FBitmap;
end;
```

Tip 14: Hide Components

Too often I hear OOP purists complaining because forms include the list of the components in the published section, an approach that doesn't conform to the principle of encapsulation. They are actually pointing out an important issue, but most of them seem to be unaware that the solution is at hand without rewriting the libraries or changing the language. The component references which are added to a form can be moved to the private portion of the form declaration, so that they won't be accessible by other forms. This way you can make compulsory use of properties mapped to the components (see the section above) to access their status.

If the IDE places all the components in the published section, this is because of the way these fields are bound to the components created from the streaming file (DFM or FMX). When you set a component's name the VCL automatically attaches the component object to its reference in the form. This is possible only if the reference is published, because the streaming system uses the traditional RTTI and some `ToObject` methods to perform the operation.

So what happens is that by moving the component references from the published to the private section you lose this automatic behavior. To fix the problem, simply

320 - 10: Properties and Events

make it manual, by adding the following code for each component in the `OnCreate` event handler of the form:

```
| Edit1 := FindComponent('Edit1') as TEdit;
```

The second operation you have to do is register the component classes in the system, so that their RTTI information is included in the compiled program and made available to the system. This is needed only once for every component class, and only if you move all the component references of this type to the private section. You can add this call even if you are not sure it is needed for your application, given that as an extra call to the `RegisterClasses` method for the same class has no effect. The `RegisterClasses` call is usually added to the initialization section of the unit hosting the form:

```
| RegisterClasses([TEdit]);
```

Tip 15: Use an OOP Form Wizard

Repeating the two operations above for every component of every form is certainly boring and time consuming. To avoid this excessive burden, I've written a simple wizard which generates the lines of code to add to the program in a small window. You'll need to do a simple copy and paste operations for each form, as the wizard doesn't automatically place the source code in the initialization section of the unit.

How to get the wizard? You can find it as part of the “Cantools Wizards” at:

```
| https://github.com/marcocantu/cantools
```

Tips Conclusion

This is only a small collection of tips and suggestions for a more balanced RAD and OOP development model. Of course there is much more to this topic, which goes well beyond the focus of this book, which is primarily on the language itself, not the best practices for application architectures.

note There are a few books covering applications architecture with Delphi, but none better than the series of volumes written by Nick Hodges, including “Coding in Delphi”, “More Coding in Delphi”, and “Dependency Injection in Delphi”. You can find more information about these books at <http://www.codingindelphi.com/>

11: interfaces

Contrary to what happens in C++ and a few other languages, the Object Pascal inheritance model doesn't support multiple inheritance. This means that each class can have only a single base class.

The usefulness of multiple inheritance is a topic of debate among OOP experts. The absence of this construct in Object Pascal can be considered a disadvantage because you don't have the power of C++ but also an advantage because you get a simpler language and avoid some of the problems multiple inheritance introduces. One way to compensate the lack of multiple inheritance in Object Pascal is the use of interfaces, which allow you to define a class that implements multiple abstractions at the same time.

note Most of today's object-oriented programming languages do not support multiple inheritance, but use interfaces, including Java and C#. Interfaces provide the flexibility and power of declaring support for multiple interfaces implemented on a class, while avoiding the problems of inheriting multiple implementations. Multiple inheritance support remains mostly confined to the C++ language. Some dynamic object-oriented languages support mix-ins, a different and simpler way to achieve something similar to multiple inheritance.

Here rather than opening up a debate, I'll simply assume that it is useful to treat a single object from multiple "perspectives". But before I build an example to explain this principle, we have to introduce the role of interfaces in Object Pascal and figure out how they work.

322 - 11: Interfaces

From a more general point of view, interfaces support a slightly different object-oriented programming model than classes. Objects implementing interfaces are subject to polymorphism for each of the interfaces they support. Indeed, the interface-based model is powerful. Interfaces favor encapsulation and provide a looser connection between classes than inheritance.

note The techniques covered in this chapter and the overall support for interfaces were originally added to the Object Pascal language as a way to support and implement Windows COM (Component Object Model) architecture. Later on, the feature was extended to be fully usable outside of that scenario, but some of the COM elements like interface identity via an ID and reference counting support still remain in the current Object Pascal implementation of interfaces, making them slightly different from most other languages.

Using Interfaces

Beside declaring abstract classes (classes with abstract methods), in Object Pascal you can also write a *purely abstract class*; that is, a sort of class with only virtual abstract methods. This is accomplished using a specific keyword, `interface`. For this reason we refer to these data types as *interfaces*.

Technically, an interface is not a class, although it may resemble one. While a class can have an instance, an interface cannot. An interface can be implemented by one or more classes, so that instances of those classes end up *supporting* or *implementing* the interface.

In Object Pascal, interfaces have a few distinctive features:

- Interface type variables are reference counted, unlike class type variables, offering a form of automatic memory management
- A class can inherit from only one base class, it can implement multiple interfaces
- Just as all classes inherit from `TObject`, all interfaces descend from `IInterface`, forming a separate, orthogonal hierarchy
- By convention interface names start with the letter I, rather than the letter T used by most other data types

note Originally the base interface type in Object Pascal was called `IUnknown`, as this is what COM requires. The `IUnknown` interface was later renamed `IInterface`, to underline the fact you can use interface in Object Pascal even outside of the COM realm and on operating systems where COM doesn't exist. Anyway, the actual behavior of `IInterface` is still identical to the previous one of `IUnknown`.

Declaring an Interface

After looking at the core concepts, let's move to some actual code that should help you understand how interfaces work in Object Pascal. In practical terms, an interface has a definition that resembles a class definition. This definition has a list of methods, but those methods are not implemented in any way, exactly as happens for an abstract method in a regular class.

The following is the definition of an interface:

```
type
  ICanFly = interface
    function Fly: string;
  end;
```

Given each interface directly or indirectly inherits from the base interface type, this corresponds to writing:

```
type
  ICanFly = interface (IInterface)
    function Fly: string;
  end;
```

In a little while I'll show you what's the implication of inheriting from `IInterface` and what it brings to the table. For the moment, suffice to say that `IInterface` has a few base methods (again, similar to `TObject`).

There is one last tidbit related to interface declarations. For interfaces, part of the type checking is done dynamically, and the system requires each interface to have a unique identifier, or GUID, that you can obtain in the Delphi editor by pressing Ctrl+Shift+G. This is the complete code of the interface:

```
type
  ICanFly = interface
    ['{D7233EF2-B2DA-444A-9B49-09657417ADB7}']
    function Fly: string;
  end;
```

This interface and its implementation (described below) are available in the `Intf101` application project.

note Although you can compile and use an interface without specifying a GUID, you'll generally want to generate the GUID because it is required to perform interface querying or dynamic as type-casts using that interface type. The whole point of interfaces is to take advantage of greatly extended type flexibly at run time, which depends on interfaces have a GUID.

Implementing the Interface

Any class can implement one or more interfaces by listing them after the base class it inherits from, and providing an implementation for each of the methods of each of the interfaces:

```
type
  TAirplane = class (... , ICanFly)
    function Fly: string;
  end;

function TAirplane.Fly: string;
begin
  // actual code
end;
```

When a class implements an interface it *must* provide an implementation of all of the interface methods with the same signature, so in this case the `TAirplane` class must implement the `Fly` method as a function returning a string. Given the interface also inherits from a base interface (`IInterface`), a class implementing an interface must invariably provide all of the methods of the interface and of all of its base interfaces.

This is why it is quite common to implement interfaces in classes that inherit from a base class that already implements the methods of the `IInterface` base interface. The Object Pascal runtime library already provides a few base classes to implement the basic behavior. The simplest one is the `TInterfacedObject` class, so the code above could become:

```
type
  TAirplane = class (TInterfacedObject, ICanFly)
    function Fly: string;
  end;
```

note When you implement an interface you can use either a static or a virtual method. If you plan overriding the methods in inherited class, using virtual methods makes sense. There is an alternative approach, though, which is to specify that the base class also inherits from the same interface, and overrides the method. I tend to prefer declaring the method that implement interface methods as virtual methods when needed.

Now that we have defined an interface and a class implementing it, we can create an object of this class. We can treat this as a regular class, by writing:

```
var
  Airplane1: TAirplane;
begin
  Airplane1 := TAirplane.Create;
  try
    Airplane1.Fly;
  finally
    Airplane1.Free;
  end;
end;
```

In this case we are ignoring the fact that the class implements an interface. The difference is that now we can also declare a variable of the interface type. Using an interface type variable *automatically* enables the reference memory model, so we can skip the try-finally block:

```
var
  Flyer1: ICanFly;
begin
  Flyer1 := TAirplane.Create;
  Flyer1.Fly;
end;
```

There are a few relevant considerations about the first line of this apparently simple code snippet, also part of the `Intf101` application project. First, as soon as you assign an object to an interface variable, the run time automatically checks to see whether the object implements that interface, using a special version of the `as` operator. You can explicitly express this operation by writing the same line of code as follows:

```
Flyer1 := TAirplane.Create as ICanFly;
```

Second, whether we use the direct assignment or the `as` statement, the runtime does one extra thing: it calls the `_AddRef` method of the object, increasing its reference count. This is done by calling the method our object inherits from the base class `TInterfacedObject`.

At the same time, as soon as the `Flyer1` variable goes out of scope (that is, when executing the `end` statement), the Delphi run time calls the `_Release` method, which decreases the reference count, checks whether the reference count is zero, and in this case destroys the object. This is why in the code above, there is no need to manually free the object we've created in code and no need to write a try-finally block.

note While the source code above has no `try-finally` block, the compiler will automatically add to the method an implicit `try-finally` block with the implicit call to `_Release`. This takes place in many cases in Object Pascal: basically every time a method has one or more managed types (like strings, interface, or dynamic arrays) there is an automatic and implicit `try-finally` block added by the compiler.

Interfaces and Reference Counting

As we have seen in the code above, Object Pascal objects referenced by interface variables are reference-counted (unless the interface type variable is marked as weak or unsafe, as explained later). We have also seen that they can be automatically deallocated when no interface variable refers to them any longer.

What is important to acknowledge is that while there is some compiler magic involved (the hidden `_AddRef` and `_Release` calls) the actual reference counting mechanism is subject to a specific implementation provided by the developer or the run-time library. In the last example, reference counting is indeed in action, because of the code in the methods of the `TInterfacedObject` class (here listed in a slightly simplified version):

```
function TInterfacedObject._AddRef: Integer;
begin
    Result := AtomicIncrement(FRefCount);
end;

function TInterfacedObject._Release: Integer;
begin
    Result := AtomicDecrement(FRefCount);
    if Result = 0 then
        begin
            Destroy;
        end;
end;
```

Now consider a different base class that implements `IInterface` and that is also found in the RTL (in the `Generics.Defaults` unit), `TSingletonImplementation`. This oddly named class basically disables the actual reference counting mechanism:

```
function TSingletonImplementation._AddRef: Integer;
begin
    Result := -1;
end;

function TSingletonImplementation._Release: Integer;
begin
    Result := -1;
end;
```

note The `TSingletonImplementation` class is a real misnomer as it has nothing to do with the singleton pattern. We'll see an example of this pattern in the next chapter.

While `TSingletonImplementation` is not commonly used, there is another class that implements interfaces and disables the reference counting mechanism, just because it has its own memory management model, and that's the `TComponent` class.

If you want to have a custom component that implements an interface, you don't have to worry about reference counting and memory management. We'll see an example of a custom component implementing an interface in the section “Implementing Patterns with Interfaces” at the end of this chapter.

Errors in Mixing References

When using objects, you should generally access them either only with object variables or only with interface variables. Mixing the two approaches breaks the reference counting scheme provided by Object Pascal and can cause memory errors that are extremely difficult to track. In practice, if you've decided to use interfaces, you should probably use exclusively interface-based variables.

Here is one example out of many possible similar cases. Suppose you have an interface, a class implementing it, and a global procedure taking the interface as parameter:

```
type
  IMyInterface = interface
    ['{F7BEADFD-ED10-4048-BB0C-5B232CF3F272}']
    procedure Show;
  end;

  TMyIntfObject = class (TInterfacedObject, IMyInterface)
  public
    procedure Show;
  end;

procedure ShowThat (AnIntf: IMyInterface);
begin
  AnIntf.Show;
end;
```

The code looks fairly trivial and it is 100 percent correct. What might be wrong is the way you call the procedure (this code is part of the `IntfError` application project):

```
procedure TForm1.BtnMixClick(Sender: TObject);
```


328 - 11: Interfaces

```
var
  AnObj: TMyIntfObject;
begin
  AnObj := TMyIntfObject.Create;
  try
    ShowThat (AnObj);
  finally
    AnObj.Free;
  end;
end;
```

What happens in this code is I'm passing a plain object to a function expecting an interface. Given the object does support the interface the compiler has no problem making the call. The issue is in the way memory is managed.

Initially, the object has a reference count of zero, as there is no interface referring to it. On entering the `ShowThat` procedure, the reference count is increased to 1. That's OK, and the call takes place. Now upon exiting the procedure the reference count is decreased and goes to zero, so the object is destroyed. In other words, the `AnObj` will be destroyed as you pass it to a procedure, which is indeed quite awkward. If you run this code, it will fail with a memory error.

There could be multiple solutions. You could artificially increment the reference count and use other low level tricks. But the real solution is to not mix interfaces and object references and go for using only interfaces to refer to an object (this code is taken again from the `IntfError` application project):

```
procedure TForm1.BtnIntfOnlyClick(Sender: TObject);
var
  AnIntf: IMyInterface;
begin
  AnIntf := TMyIntfObject.Create;
  ShowThat (AnIntf);
end;
```

In this specific case the solution is at hand, but in many other circumstances it is very hard to come up with the correct code. Again, the rule of the thumb is to avoid mixing references of different types... but keep also reading the next section for some recent alternative approaches.

Weak And Unsafe Interface References

Starting with Delphi 10.1 Berlin, the Object Pascal language offers some enhancements on the management of interface references. The language in fact offers different type of references:

- Regular references increment and decrement the object reference count when assigned and released, eventually freeing the object when the reference count reaches zero
- Weak references (marked with the `[weak]` modifier) do not increase the reference count of the object they refer to. These references are fully managed, so they are automatically set to `nil` if the referenced object is destroyed.
- Unsafe references (marked with the `[unsafe]` modifier) do not increase the reference count of the object they refer to and are not managed – something not much different from a basic pointer.

note The use of weak and unsafe references was originally introduced as part of the ARC memory management support for mobile platforms. As ARC is now phased out, this feature remains available only for interface references.

In the common scenarios in which reference count is active, you can have code like the following, which relies on reference counting to dispose the temporary object:

```
procedure TForm3.Button2Click(Sender: TObject);
var
    OneIntf: ISimpleInterface;
begin
    OneIntf := TObjectOne.Create;
    OneIntf.DoSomething;
end;
```

What if the object has a standard reference count implementation and you want to create an interface reference that is kept out of the total count of references? You can now achieve this by adding the `[unsafe]` attribute to the interface variable declaration, changing the code above to:

```
procedure TForm3.Button2Click(Sender: TObject);
var
    [unsafe] OneIntf: ISimpleInterface;
begin
    OneIntf := TObjectOne.Create;
    OneIntf.DoSomething;
end;
```

Not that this is a good idea, as the code above would cause a memory leak. By disabling the reference counting, when the variable goes out of scope nothing happens. There are some scenarios in which this is beneficial, as you can still use interfaces and not trigger the extra reference. In other words, an unsafe reference is treated just like... a pointer, with no extra compiler support.

Now before you consider using the unsafe attribute for having a reference without increasing the count, consider that in most cases there is another better option, that is the use of weak references. Weak references also avoid increasing the reference

330 - 11: Interfaces

count, but they are managed. This means that the system keeps track of weak references, and in case the actual object gets deleted, it will set the weak reference to `nil`. With an unsafe reference, instead, you have no way to know the status of the target object (a scenario called dangling reference).

In which scenarios are weak reference useful? A classic case is that of two object with cross-references. In such a case, in fact, the object would artificially inflate the reference count of the other objects, and they'll basically remain in memory forever (with reference count set to 1), even when they become unreachable.

As an example consider the following interface, accepting a reference to another interface of the same time, and a class implementing it with an internal reference:

```
type
  ISimpleInterface = interface
    procedure DoSomething;
    procedure AddObjectRef (Simple: ISimpleInterface);
  end;

  TObjectOne = class (TInterfacedObject, ISimpleInterface)
  private
    AnotherObj: ISimpleInterface;
  public
    procedure DoSomething;
    procedure AddObjectRef (Simple: ISimpleInterface);
  end;
```

If you create two objects and cross-reference them, you end up with a memory leak:

```
var
  One, Two: ISimpleInterface;
begin
  One := TObjectOne.Create;
  Two := TObjectOne.Create;
  One.AddObjectRef (Two);
  Two.AddObjectRef (One);
```

Now the solution available in Delphi is to mark the private field `AnotherObj` as a weak interface reference:

```
private
  [weak] AnotherObj: ISimpleInterface;
```

With this change, the reference count is not modified when you pass the object as parameter to the `AddObjectRef` call, it stays at 1, and it goes back to zero when the variables go out of scope, freeing the objects from memory.

Now there are many other cases in which this feature becomes handy, and there is some real complexity in the underlying implementation. It is great feature, but one that takes some effort to fully master. Also, it does have some run-time cost, as weak references are managed (while unsafe ones are not).

For additional information about weak references for interfaces and how they work you can refer to the section “Memory Management and Interfaces:” in Chapter 13, “Objects and Memory”.

Advanced Interface Techniques

To further delve into the capabilities of interfaces, before we look into real world usage scenarios, it is important to cover some of their more advanced technical features, such as how to implement multiple interfaces or how to implement an interface method with a method having a different name (in case of a name conflict).

Another important feature interfaces have are properties. To demonstrate all of these more advanced features related with interfaces, I’ve written the `IntfDemo` application project.

Interface Properties

The code in this section is based on two different interfaces, `IWalker` and `IJumper`, both of which define a few methods and a property. An interface property is just a name mapped to a `read` and a `write` method. Unlike a class, you cannot map an interface property to a field, simply because an interface cannot have a field.

These are the actual interface definitions:

```
IWalker = interface
  ['{0876F200-AAD3-11D2-8551-CCA30C584521}']
  function Walk: string;
  function Run: string;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;

  property Position: Integer
    read GetPos write SetPos;
end;

IJumper = interface
  ['{0876F201-AAD3-11D2-8551-CCA30C584521}']
  function Jump: string;
  function Walk: string;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;
```

332 - 11: Interfaces

```
    property Position: Integer
        read GetPos write SetPos;
end;
```

When you implement an interface with a property, all you have to implement is the actual access methods, as the property is transparent and not available in the class itself:

```
TRunner = class (TInterfacedObject, Iwalker)
private
    FPos: Integer;
public
    function walk: string;
    function Run: string;
    procedure SetPos (Value: Integer);
    function GetPos: Integer;
end;
```

The implementation code is not complex (you can find it in the `IntfDemo` application project), with methods computing the new position and displaying what's being executed:

```
function TRunner.Run: string;
begin
    Inc (FPos, 2);
    Result := FPos.ToString + ' : Run';
end;
```

The demo code using the `Iwalker` interface and its `TRunner` implementation is this:

```
var
    Intf: Iwalker;
begin
    Intf := TRunner.Create;
    Intf.Position := 0;
    Show (Intf.walk);
    Show (Intf.Run);
    Show (Intf.Run);
end;
```

The output should not be surprising:

```
1: walk
3: Run
5: Run
```

Interface Delegation

In a similar way, I can define a simple class implementing the `IJumper` interface:

```
TJumperImpl = class (TAggregatedObject, IJumper)
private
    FPos: Integer;
```

```

public
  function Jump: string;
  function Walk: string;
  procedure SetPos (Value: Integer);
  function GetPos: Integer;
end;

```

This implementation is different from the previous one for the use of a specific base class, `TAggregatedObject`. This is a special purpose class for the definition of the objects used internally to support an interface, with the syntax I'm going to show in a moment.

note The `TAggregatedObject` class is another implementation of `IInterface` defined in the `System` unit. Compared to `TInterfacedObject` it has differences in the implementation of reference counting (basically delegating all reference counting to the *container* or *controller*) and in the implementation of interface querying, in case the container supports multiple interfaces.

I'm going to use it in a different way. In the following class, `TMyJumper`, I don't want to repeat the implementation of the `IJumper` interface with similar methods. Instead, I want to *delegate* the implementation of that interface to a class already implementing it. This cannot be done through inheritance (we cannot have two base classes); instead, you can use specific features of the language - interface delegation. The following class implements an interface by referring to an implementation object with a property, rather than implementing the actual methods of the interface:

```

TMyJumper = class (TInterfacedObject, IJumper)
private
  FJumpImpl: TJumperImpl;
public
  constructor Create;
  destructor Destroy; override;
  property Jumper: TJumperImpl
    read FJumpImpl implements IJumper;
end;

```

This declaration indicates that the `IJumper` interface is implemented for the `TMyJumper` class by the `FJumpImpl` field. This field, of course, must actually implement all the methods of the interface. To make this work, you need to create a proper object for the field when a `TMyJumper` object is created (the constructor parameter is required by the base `TAggregatedObject` class):

```

constructor TMyJumper.Create;
begin
  FJumpImpl := TJumperImpl.Create (self);
end;

```

334 - 11: Interfaces

The class has also a destructor for freeing the internal object, which is referenced with a regular field and not an interface (as reference counting won't work in this scenario).

This example is simple, but in general, things get more complex as you start to modify some of the methods or add other methods that still operate on the data of the internal `FJumpImpl` object. The overall concept here is that you can reuse the implementation of an interface in multiple classes.

The code that uses this interface implemented indirectly is identical to the standard code one would write:

```
procedure TForm1.Button2Click(Sender: TObject);  
var  
    Intf: IJumper;  
begin  
    Intf := TMyJumper.Create;  
    Intf.Position := 0;  
    Show (Intf.Walk);  
    Show (Intf.Jump);  
    Show (Intf.Walk);  
end;
```

Multiple Interfaces and Methods Aliases

Another very important features of interfaces is the ability for a class to implement more than one. This is demonstrated by the following the `TAthlete` class, which implements both the `IWalker` and `IJumper` interfaces:

```
TAthlete = class (TInterfacedObject, IWalker, IJumper)  
private  
    FJumpImpl: TJumperImpl;  
public  
    constructor Create;  
    destructor Destroy; override;  
    function Run: string; virtual;  
    function walk1: string; virtual;  
    function IWalker.Walk = walk1;  
    procedure SetPos (Value: Integer);  
    function GetPos: Integer;  
  
    property Jumper: TJumperImpl  
        read FJumpImpl implements IJumper;  
end;
```

One of the interfaces is implemented directly, whereas the other is delegated to the internal `FJumpImpl` object, exactly as I did in the previous example.

Now we have an issue. Both the interfaces we want to implement have a `walk` method, with the same parameters signature, so how to we implement both of them in our class? How does the language support a method name clash, in case of multiple interfaces? The solution is to give the method a different name and map it to the specific interface method using it as a prefix, with the statement:

```
function Iwalker.walk = walk1;
```

This declaration indicates that the class implements the `walk` method of the `Iwalker` interface with a method called `walk1` (instead of with a method having the same name). Finally, in the implementation of all of the methods of this class, we need to refer to the `Position` property of the `FJumpImpl` internal object.

By declaring a new implementation for the `Position` property, we'll end up with two positions for a single athlete, a rather odd situation. Here are a couple of examples:

```
function TAthlete.GetPos: Integer;
begin
    Result := FJumpImpl.Position;
end;

function TAthlete.Run:string;
begin
    FJumpImpl.Position := FJumpImpl.Position + 2;
    Result := IntToStr (FJumpImpl.Position) + ': Run';
end;
```

How can we create an interface to this `TAthlete` object and refer to both operations in the `Iwalker` and `IJumper` interfaces? Well, we cannot do that exactly, because there isn't a base interface we can use. Interfaces allow for a more dynamic type checking and type casting, though, so what we can do is convert an interface to a different one, as long as the object we are referring to supports both interfaces, something the compiler will be able to find out only at run-time. This is the code for that scenario:

```
procedure TForm1.Button3Click(Sender: TObject);
var
    Intf: Iwalker;
begin
    Intf := TAthlete.Create;
    Intf.Position := 0;
    Show (Intf.Walk);
    Show (Intf.Run);
    Show ((Intf as IJumper).Jump);
end;
```

Of course, we could have picked either of the two interfaces, and converted that to the other. Using an `as` cast is a way to do a run-time conversion, but there are more options when you are dealing with interfaces, as we'll see in the next section.

Interface Polymorphism

In the previous section we have seen how you can define multiple interfaces and have a class implement two of them. Of course, this could be extended to any number. You can also create a hierarchy of interfaces, as an interface can inherit from an existing interface:

```
ITripleJumper = interface (IJumper)
  [ '{0876F202-AAD3-11D2-8551-CCA30C584521}' ]
  function TripleJump: string;
end;
```

This new interface type has all of the methods (and properties) of its base interface type and adds a new method. Of course, there are interface compatibility rules much the same as classes.

What I want to focus on in this section, though, is a slightly different topic, which is interface-based polymorphism. Given a general base class object, we can invoke a virtual method and be sure the correct version will be called. The same can happen for interfaces. With interfaces, however, we can go a step beyond and often have dynamic code that queries for an interface. Given that the object to interface relationship can be quite complex (an object can implement multiple interfaces and indirectly implement also their base interfaces) it is important to have a better picture of what's possible in this scenario.

As a starting point, let's suppose we have a generic `IInterface` reference. How do we know if it supports a specific interface? There are actually multiple techniques, which are slightly different from their class counterparts:

- Use an `is` statement for testing (and possibly an `as` cast for the following conversion). This can be used to check if an object supports an interface, but not if an object referenced with an interface also supports another one (that is, you cannot apply `is` to interfaces). Notice that in any case the `as` conversion is required: a direct cast to an interface type will almost invariably result in an error.
- Call the global `Support` function, using one of its many overloaded versions. You can pass to this function the object or the interface to test, the target interface (using the GUID or the type name), and you can also pass an interface variable where the actual interface will get stored, if the function is successful.
- Directly call the `QueryInterface` method of the `IInterface` base interface, which is a little lower level, always requires an interface type variable as an extra result, and uses a numeric `HRESULT` value rather than a Boolean result.

Here is a snippet taken from the same `IntfDemo` application project showing how you can apply the last two techniques to a generic `IInterface` variable:

```

procedure TForm1.Button4Click(Sender: TObject);
var
    Intf: IInterface;
    walkIntf: Iwalker;
begin
    Intf := TAthlete.Create;
    if Supports (Intf, Iwalker, walkIntf) then
        Show (walkIntf.Walk);

    if Intf.QueryInterface (Iwalker, walkIntf) = S_OK then
        Show (walkIntf.Walk);
end;

```

I fully recommend using one of the overloaded versions of the `Supports` function, compared to the `QueryInterface` call. Ultimately, `Supports` will call it but offers simpler, higher level options.

Another case in which you'd want to use polymorphism with interfaces, is when you have an array of interfaces of a higher level interface type (but also possibly an array of objects, some of which might support the given interface).

Extracting Objects from Interface References

It was the case for many versions of Object Pascal, that when you had assigned an object to an interface variable, there was no way to access the original object. At times, developers would add a `GetObject` method to their interfaces to perform the operation, but that was quite an odd design.

In today's language you can cast interface references back to the original object to which they have been assigned. There are three separate operations you can use:

- You can write an `is` test to verify that an object of the given type can indeed be extracted from the interface reference:

```

| IntfVar is TMyObject

```

- You can write an `as` cast to perform the type cast, raising an exception in case of an error:

```

| IntfVar as TMyObject.

```

- You can write a hard type cast to perform the same conversion, returning a `nil` pointer in case of an error:

```

| TMyObject(IntfVar)

```

note In every case, the type cast operation works only if the interface was originally obtained from an Object Pascal object, and not from a COM server. Also note that you can not only cast to the exact class of the original object, but also to one of its base classes (following type compatibility rules).

338 - 11: Interfaces

As an example, consider having the following simple interface and implementation class (part of the `ObjFromIntf` application project):

```
type
  ITestIntf = interface (IInterface)
    ['{2A77A244-DC85-46BE-B98E-A9392EF2A7A7}']
    procedure DoSomething;
  end;

  TTestImpl = class (TInterfacedObject, ITestIntf)
  public
    procedure DoSomething;
    procedure DoSomethingElse; // not in the interface
    destructor Destroy; override;
  end;
```

With these definitions you can now define an interface variable, assign an object to it, and use it also to invoke the method not in the interface, with the new cast:

```
var
  Intf: ITestIntf;
begin
  Intf := TTestImpl.Create;
  Intf.DoSomething;
  (Intf as TTestImpl).DoSomethingElse;
```

You can also write the code in the following way, using an `is` test and a direct cast, and you can always cast to a base class of the actual class of the object:

```
var
  Intf: ITestIntf;
  Original: TObject;
begin
  Intf := TTestImpl.Create;
  Intf.DoSomething;
  if Intf is TObject then
    Original := TObject (Intf);
  (Original as TTestImpl).DoSomethingElse;
```

Considering that a direct cast returns `nil` if not successful, you could also write the code as follows (without the `is` test):

```
Original := TObject (intf);
if Assigned (Original) then
  (Original as TTestImpl).DoSomethingElse;
```

Notice that assigning the object extracted from the interface to a variable exposes you to reference counting issues: when the interface is set to `nil` or goes out of scope, the object is actually deleted and the variable referring to it will become invalid. You'll find the code highlighting the problem in the `BtnRefCountIssueClick` event handler of the example.

Implementing An Adapter Pattern with Interfaces

As a real world example of the use of interfaces, I've added to this chapter a section covering the adapter pattern. In short, the adapter pattern is used to convert the interface of one class into another one expected by the user of the class. This allows you to use an existing class within a framework requiring a defined interface.

The pattern can be implemented by creating a new class hierarchy that does the mapping, or by extending existing classes so that they expose a new interface. This can be done either by multiple inheritance (in the languages supporting it) or using interfaces. In this last case, which is what I'm going to use here, a new inherited class will implement the given interface and map to its method its existing behavior.

In the specific scenario, the adapter provides a common interface for querying values of multiple components, which happen to have inconsistent interfaces (as it often happens in the UI libraries). This is the interface, called `ITextAndValue` because it allows accessing the status of a component by getting either a textual description or a numeric one:

```
type
  ITextAndValue = interface
    '[51018CF1-0D3C-488E-81B0-0470B09013EB]'
    procedure SetText(const Value: string);
    procedure SetValue(const Value: Integer);
    function GetText: string;
    function GetValue: Integer;

    property Text: string read GetText write SetText;
    property Value: Integer read GetValue write SetValue;
end;
```

The next step is to create a new subclass for each of the components we want to be able to use with the interface. For example, we could write:

```
type
  TAdapterLabel = class(TLabel, ITextAndValue)
  protected
    procedure SetText(const Value: string);
    procedure SetValue(const Value: Integer);
    function GetText: string;
    function GetValue: Integer;
end;
```

The implementation of these four methods is quite simple, as they can be mapped to the `Text` property performing a type conversion in case the value (or the text) is a number. Now that you have a new component, however, you'll have to install it (as

340 - 11: Interfaces

we mentioned in the last chapter) and replace the existing components in your forms with this new one. Repeating the same process for each of the components you want to *adapt* would be very time consuming.

A much simpler alternative would be to use the *interposer class idiom* (that is, define a class with the same name of the base class, but in a different unit). This will be properly recognized by the compiler and by the run-time streaming system, so that at runtime you'll end up with an object of the new specific class. The only difference is that at design time you'll see and interact with instances of the base component class.

note Interposer classes were first mentioned, and given this name, many years ago in The Delphi Magazine. They certainly are a bit of a hack, but at times a handy one. I consider interposer classes, that is classes with the same name of a base class but defined in a different unit, more of an Object Pascal idiom. Notice that for this mechanism to work it is critical that the unit with the interposer class is listed in the uses statement after the one with the regular class it should replace. In other words, the symbols defined in the last units in the uses statement replace an identical symbol define in previously included units. Of course, you can always discriminate the symbols by prefixing them with the unit name, but this would really defeat the entire idea of this hack, which is taking advantage of the global name resolution rules.

To define an interposer class, you'd generally write a new unit with a class having the same name of an existing base class. To refer to the base class, you have to prefix it with the unit name (or the compiler will consider it a recursive definition):

```
type
  TLabel = class(StdCtrls.TLabel, ITextAndValue)
  protected
    procedure SetText(const value: string);
    procedure SetValue(const value: Integer);
    function GetText: string;
    function GetValue: Integer;
  end;
```

In this case you won't have to install components or touch the existing programs, but only add an extra uses statement to them at the end of the list. In both cases (but the demo application I wrote uses interposer classes), you can query the components of the form for this adapter interface and, for example, write code to set all of the *values* to 50, which will affect different properties of different components:

```
var
  Intf: ITextAndValue;
  I: integer;
begin
  for I := 0 to ComponentCount - 1 do
    if Supports (Components [I], ITextAndValue, Intf) then
      Intf.Value := 50;
end;
```

In the specific example, this code will affect the `value` of a progress bar or a number box, and the `Text` of a label or an edit. It will also totally ignore a couple of other components for which I didn't define the adapter interface. While this is just a very specific case, if you examine other design patterns you'd easily find out that quite a few of them can be better implemented taking advantage of the extra flexibility interfaces have over classes in Object Pascal (like in Java and C#, just to name another couple of popular languages that make extensive use of interfaces).

342 - 11: Interfaces

12: manipulating classes

In the last few chapters you've seen the foundations of the object side of the Object Pascal language: classes, objects, methods, constructors, inheritance, late binding, interfaces, and more. Now we need to move one step further, by looking at some more advanced and rather specific features of the language related to managing classes. From class references to class helpers, this chapter covers many features not found in other OOP languages, or at least implemented significantly differently.

The focus is classes, and manipulating classes at run-time, a topic we'll further augment when covering reflection and attributes in Chapter 16.

Class Methods and Class Data

When you define a class in Object Pascal and most other OOP language, you define the data structure of the objects (or instances) of the class and the operations that you can perform on such an object. There is also the possibility, however, to define

344 - 12: Manipulating Classes

data shared among all objects of the class and methods that can be called for the class independently from any actual object created from it.

To declare a class method in Object Pascal, you simply add the `class` keyword in front of it and you can see this for both procedures and functions:

```
type
  TMyClass = class
    class function ClassMeanValue: Integer;
```

Given an object `MyObject` of class `TMyClass`, you can call the method either by applying it to an object or to the class as a whole:

```
var
  MyObject: TMyClass;
begin
  ...
  I := TMyClass.ClassMeanValue;
  J := MyObject.ClassMeanValue;
```

This syntax implies you can call the class method even before an object of the class has been created. There are scenarios of classes made only of class methods, with the implicit idea that you'll never create an objects of these classes (something you can enforce by declaring the `Create` constructor private).

note The use of class methods in general and of classes made only of class methods in particular is more common in OOP languages that don't allow the use of global functions. Object Pascal still let's you declare *old-fashioned* global functions, but over recent years the system libraries and the code written by developers has moved more and more towards a consistent use of class methods. The advantage of using class methods is that they become logically tied to a class, which acts as a sort of name space for a group of related functions.

Class Data

Class data is data shared among all objects of the class, offering global storage but class specific access (including access limitations). How do you declare class data? Simply by defining a new section of the class marked with the `class var` keyword combination:

```
type
  TMyData = class
    private
      class var
        CommonCount: Integer;
    public
      class function GetCommon: Integer;
```

The `class var` section introduces a block of one or more declarations. You can use a `var` section (which is a new way to use this keyword) to declare other instance fields in the same section (private below):

```
type
  TMyData = class
    private
      class var
        CommonCount: Integer;
      var
        MoreObjectData: string;
    public
      class function GetCommon: Integer;
```

In addition to declaring class data, you can also define class properties, as we'll see in a later section.

Virtual Class Methods and the Hidden Self Parameter

While the concept of class methods is shared among programming languages, the Object Pascal implementation has a few peculiarities. First, class methods have an implicit (or hidden) `self` parameter, much like instance method. However, this hidden `self` parameter is a reference to the class itself, not to an instance of the class.

At first sight, the fact that a class method has a hidden parameter that refers to the class itself might seem quite useless. The compiler knows the class of a method, after all. However, there is a peculiar language feature that explains this: Unlike most other languages, in Object Pascal class methods can be virtual. In a derived class, you can override a base type class method, like you can do for a regular method.

note The support for virtual class method is connected with the support for virtual constructors (which are some sort of special purpose class methods). Both features are not found in many compiled and strongly typed OOP languages.

Class Static Methods

Class static methods have been introduced in the language for platform compatibility. The differences between ordinary class methods and class static methods are that class static methods have no references to their own class (no `self` parameter indicating the class itself) and cannot be virtual.

346 - 12: Manipulating Classes

Here is a simple example with some incorrect statements commented out, taken from the ClassStatic application project:

```
type
  TBase = class
  private
    Tmp: Integer;
  public
    class procedure One;
    class procedure Two; static;
    ...
  end;

class procedure TBase.One;
begin
  // Error: Instance member 'Tmp' inaccessible here
  // Show (Tmp);
  Show ('one');
  Show (self.ClassName);
end;

class procedure TBase.Two;
begin
  Show ('two');
  // error: Undeclared identifier: 'self'
  // Show (self.ClassName);
  Show (ClassName);
  Two;
end;
```

In both cases you can call these class methods directly or invoke them through an object:

```
TBase.One;
TBase.Two;

Base := TBase.Create;
Base.One;
Base.Two;
```

There are two interesting features that make class static methods useful in Object Pascal. The first is that they can be used to define class properties, as described in a coming section. The second is that class static methods are fully C-language compatible, as explained below.

Static Class Methods and Windows API Callbacks

The fact they have no hidden self parameter implies static class methods can be passed to the operating system (for example, on Windows) as callback functions. In practice, you can declare a static class method with the `stdcall` calling convention

and use it as a direct Windows API callback, as I've done for the `TimerCallback` method of the `StaticCallback` application project:

```
type
  TFormCallback = class(TForm)
    ListBox1: TListBox;
    procedure FormCreate(Sender: TObject);
  private
    class var
      NTimerCount: Integer;
  public
    class procedure TimerCallback (hwnd: THandle;
      uMsg, idEvent, dwTime: Cardinal); static; stdcall;
  end;
```

The class data is used by the callback as a counter. The `onCreate` handler calls the `SetTimer` API passing the address of the class static procedure:

```
procedure TFormCallback.FormCreate(Sender: TObject);
var
  Callback: TFNTimerProc;
begin
  NTimerCount := 0;
  Callback := TFNTimerProc(@TFormCallback.TimerCallback);
  SetTimer(Handle, TIMERID, 1000, Callback);
end;
```

note The parameter to `TFNTimerProc` is a method pointer which is why the name of the class static method is preceded by an `@` or by using the `Addr` function. That's because we need to get the method address, not execute the method.

Now the actual callback function increases the timer and updates the form, referring to it using the corresponding global variable, as a class method cannot refer to the form as self:

```
class procedure TFormCallback.TimerCallback(
  hwnd: THandle; uMsg, idEvent, dwTime: Cardinal);
begin
  try
    Inc (NTimerCount);
    FormCallback.ListBox1.Items.Add (
      IntToStr (NTimerCount) + ' at ' + TimeToStr(Now));
  except on E: Exception do
    Application.HandleException(nil);
  end;
end;
```

The `try-except` block is there to avoid any exception being sent back to Windows... a rule you have to use follow consistently for callback or DLL functions.

Class Properties

One of the reasons for using class static methods is to implement class properties. What is a class property? Like a standard property it is a symbol attached to read and write mechanisms. Unlike a standard property it relates to the class and must be implemented using either class data or class static methods. The `TBase` class (again, from the `ClassStatic` application project) has two class properties defined in the two different ways:

```
type
  TBase = class
  private
    class var
      FMyName: string;
  public
    class function GetMyName: string; static;
    class procedure SetMyName (Value: string); static;
    class property MyName: string read GetMyName write SetMyName;
    class property DirectName: string read FMyName write FMyName;
  end;
```

A Class with an Instance Counter

Class data and class methods can be used to hold information regarding a class as a whole. An example of this type of information could be the number of instances of the class that have been created so far... minus those already destroyed.

The `CountObj` application project shows this scenario. The program is not terribly useful, given I preferred to focus only on the specific problem and its solution. In other words, the target object has a very simple class, just storing a numeric value:

```
type
  TCountedObj = class (TObject)
  private
    FValue: Integer;
  private class var
    FTotal: Integer;
    FCurrent: Integer;
  public
    constructor Create;
    destructor Destroy; override;
    property Value: Integer read FValue write FValue;
  public
    class function GetTotal: Integer;
    class function GetCurrent: Integer;
  end;
```

Every time an object is created, the program increments both counters after calling the constructor of the base class (if any). Every time an object is destroyed, the counter current objects counter is decreased:

```
constructor TCountedObj.Create (AOwner: TComponent);
begin
    inherited Create;
    Inc (FTotal);
    Inc (FCurrent);
end;

destructor TCountedObj.Destroy;
begin
    Dec (FCurrent);
    inherited Destroy;
end;
```

The class information can be accessed without having to refer to a specific object. In fact, it might as well be that at a given time there are no objects in memory:

```
class function TCountedObj.GetTotal: Integer;
begin
    Result := FTotal;
end;
```

You can display the current status with code like:

```
Label1.Text := TCountedObj.GetCurrent.ToString + '/' +
    TCountedObj.GetTotal.ToString;
```

In the demo, this is executed in a timer, which updates a label, so it doesn't need to refer to any specific object instance nor it is triggered directly by any manual operation. The buttons in the application project, instead, just create and free some of the objects... or leave some around in memory (in fact the program has some potential memory leaks).

Class Constructors (and Destructors)

Class constructors offer a way to initialize data that relates to a class, and they have the role of class initializers, as they really don't end up *constructing* anything. A class constructor has nothing to do with a standard instance constructor: It is merely code used to initialize the class itself once, before the class is used. For example, a class constructor can set initial values for class data, load configuration or support files for the class, and so on.

In Object Pascal a class constructor is an alternative to the unit initialization code. In case both exist (in a unit), the class constructor will be executed first and then the

350 - 12: Manipulating Classes

unit initialization code. Oppositely, you can define a class destructor that will be executed after the finalization code.

A significant difference, however, is that while the unit initialization code is invariably executed if the unit is compiled in the program, the class constructor and destructor are linked only if the class is actually used. This means that the use of class constructor is much more *linker friendly* than the use of initialization code.

note In other words, with class constructors and destructors, if the type is not linked the initialization code is not part of the program and not executed; in the traditional case the opposite is true, the initialization code might even cause the linker to bring in some of the class code, even if it is never actually used anywhere else. In practical terms, this was introduced along with the gesturing framework, a rather large amount of code that is not compiled into the executable if it is not used.

In coding terms, you can write the following (see the `ClassCtor` application project):

```
type
  TTestClass = class
  public
    class var
      StartTime: TDateTime;
      EndTime: TDateTime;
    public
      class constructor Create;
      class destructor Destroy;
  end;
```

The class has two class data fields, initialized by the class constructor, and modified by a class destructor, while the initialization and finalization sections of the unit uses these data fields:

```
class constructor TTestClass.Create;
begin
  StartTime := Now;
end;

class destructor TTestClass.Destroy;
begin
  EndTime := Now;
end;

initialization
  ShowMessage (TimeToStr (TTestClass.StartTime));

finalization
  ShowMessage (TimeToStr (TTestClass.EndTime));
```

What happens is that the start up sequence works as expected, with the class data already available as you show the information. When closing, instead, the `ShowMessage` call is executed before the value is assigned by the class destructor.

Notice that you can give the class constructor and destructor any name, although `Create` and `Destroy` would be very good defaults. You cannot, however, define multiple class constructors or destructors. If you try, the compiler will issue the following error message:

```
[DCC Error] ClassCtorMainForm.pas(34): E2359 Multiple class
constructors in class TTestClass: Create and Foo
```

Class Constructors in the RTL

There are a few RTL classes that already take advantage of this language feature, like the `Exception` class that defines both a class constructor (with the code below) and a class destructor:

```
class constructor Exception.Create;
begin
  InitExceptions;
end;
```

The `InitExceptions` procedure was previously called in the initialization section of the `System.SysUtils` unit.

In general, I think that using class constructors and destructors is better than using unit initialization and termination. In most cases, this is only syntactic sugar, so you might not want to go back and change existing code. However, if you face the risk of initializing data structures you'll never used (because no class of that type is ever created) moving to class constructors will provide a definitive advantage. This is clearly more often the case in a general library, of which you don't use all of the feature, than of application code.

note A very specific case of the use of class constructors is in case of generic classes. I'll cover it in the chapter focused on generics.

Implementing the Singleton Pattern

There are classes for which it makes sense to create one and only one single instance. The Singleton pattern (another very common design pattern) requires this and also suggests having a *global point of access* for this object.

The singleton pattern can be implemented in many ways, but a classic approach is to call the function used to access the only instance exactly as `Instance`. In many cases, the implementation also follows the lazy initialization rule, so that this global instance is not created when the program starts but only the first time it is needed.

352 - 12: Manipulating Classes

In the implementation below I took advantage of class data, class methods, but also a class destructors for the final clean up. Here is the relevant code:

```
type
  TSingleton = class(TObject)
  public
    class function Instance: TSingleton;
  private
    class var TheInstance: TSingleton;
    class destructor Destroy;
  end;

class function TSingleton.Instance: TSingleton;
begin
  if TheInstance = nil then
    TheInstance := TSingleton.Create;
  Result := TheInstance;
end;

class destructor TSingleton.Destroy;
begin
  FreeAndNil (TheInstance);
end;
```

You can grab the single instance of the class (regardless of the fact this has already been created or not) by writing:

```
var
  ASingle: TSingleton;
begin
  ASingle := TSingleton.Instance;
```

Furthermore, you can hide the regular class constructor, declaring it private, so that it will be very difficult to create an object of the class without following the pattern.

Class References

Having looked at several topics related to methods, we can now move on to the topic of *class references* and further extend our example of dynamically creating components. The first point to keep in mind is that a class reference isn't a class, it isn't an object, and it isn't a reference to an object; it is simply a reference to a class type.

A class reference type determines the type of a class reference variable. Sounds confusing? A few lines of code might make this clearer. Suppose you have defined the class `TMyClass`. You can now define a new class reference type, related to that class:

```
type
  TMyClassRef = class of TMyClass;
```

Now you can declare variables of both types. The first variable refers to an object, the second to a class:

```
var
  AClassRef: TMyClassRef;
  AnObject: TMyClass;
begin
  AClassRef := TMyClass;
  AnObject := TMyClass.Create;
```

You may wonder what class references are used for. In general, class references allow you to manipulate a class data type at run time. You can use a class reference in any expression where the use of a data type is legal. Actually, there are not many such expressions, but the few cases are interesting. The simplest case is the creation of an object. We can rewrite the two lines above as follows:

```
AClassRef := TMyClass;
AnObject := AClassRef.Create;
```

This time I've applied the `Create` constructor to the class reference instead of to an actual class; I've used a class reference to create an object of that class.

note Class references are related with the concept of *metaclass* available in other OOP languages. In Object Pascal, however, a class reference is not itself a class, but only a specific type defining a reference to class data. Therefore, the analogy with *metaclasses* (classes describing other classes) is a little misleading. Actually, `TMetaClass` is also the term used in C++Builder.

When you have a class reference you can apply to it the class methods of the related class. So if `TMyClass` had a class method called `Foo`, you'd be able to write either:

```
TMyClass.Foo
AClassRef.Foo
```

That wouldn't be terribly useful, if class references didn't support the same type-compatibility rule that applies to class types. When you declare a class reference variable, such as `MyClassRef` above, you can then assign to it that specific class and any subclass. So if `MyNewClass` is a subclass of my class, you can also write

```
AClassRef := MyNewClass;
```

Now to understand why this can indeed be interesting you have to remember that the class methods you can call for a class reference can be virtual, so the specific subclass can override them. Using class references and virtual class methods you can implement a form of polymorphism at the class method level that few (if any) of the other static OOP languages support. Consider also that each class inherits from `TObject`, so you can apply to each class reference some of the methods of `TObject`, including `InstanceSize`, `ClassName`, `ParentClass`, and `InheritsFrom`. I'll discuss these class methods and other methods of `TObject` class in Chapter 17.

Class References in the RTL

The `System` unit and other core RTL units declare a lot of class references, including the following:

```
TClass = class of TObject;
ExceptClass = class of Exception;
TComponentClass = class of TComponent;
TControlClass = class of TControl;
TFormClass = class of TForm;
```

In particular, the `TClass` class reference type can be used to store a reference to any class you write in Object Pascal, because every class is ultimately derived from `TObject`. The `TFormClass` reference, instead, is used in the source code of the default Object Pascal project based on FireMonkey or the VCL. The `CreateForm` method of the `Application` object of both libraries, in fact, requires as parameter the class of the form to create:

```
Application.CreateForm(TForm1, Form1);
```

The first parameter is a class reference, the second is a variable that will receive a reference to the created object instance.

Creating Components Using Class References

What is the *practical* use of class references in Object Pascal? Being able to manipulate a data type at run time is a fundamental element of the environment. When you add a new component to a form by selecting it from the Component Palette, you select a data type and create an object of that data type. (Actually, that is what the development environment does for you behind the scenes.)

To give you a better idea of how class references work, I've built an application project called `CLASSRef`. The form displayed by this example is quite simple. It has three radio buttons, placed inside a panel in the upper portion of the form. When you select one of these radio buttons and click the form, you'll be able to create new components of the three types indicated by the button labels: radio buttons, regular push buttons, and edit boxes. For this program to run properly, it needs to change the names of the three components. The form must also have a class reference field:

```
private
    FControlType: TControlClass;
    FControlNo: Integer;
```

The first field stores a new data type every time the user clicks one of the three radio buttons, changing its status. Here is one of the three methods:

```

procedure TForm1.RadioButtonRadioChange(Sender: TObject);
begin
    FControlType := TRadioButton;
end;

```

The other two radio buttons have `OnChange` event handlers similar to this one, assigning the value `TEdit` or `TButton` to the `FControlType` field. A similar assignment is also present in the handler of the `OnCreate` event of the form, used as an initialization method. The interesting part of the code is executed when the user clicks on a `Layout` control that covers most of the surface of the form. I've chosen the `OnMouseDown` event of the form to hold the position of the mouse click:

```

procedure TForm1.Layout1MouseDown(Sender: TObject;
    Button: TMouseButton; Shift: TShiftState; X, Y: Single);
var
    NewCtrl: TControl;
    NewName: String;
begin
    // create the control
    NewCtrl := FControlType.Create (Self);

    // hide it temporarily, to avoid flickering
    NewCtrl.Visible := False;

    // set parent and position
    NewCtrl.Parent := Layout1;
    NewCtrl.Position.X := X;
    NewCtrl.Position.Y := Y;

    // compute the unique name (and text)
    Inc (FControlNo);
    NewName := FControlType.ClassName + FControlNo.ToString;
    Delete (NewName, 1, 1);
    NewCtrl.Name := NewName;

    // now show it
    NewCtrl.Visible := True;
end;

```

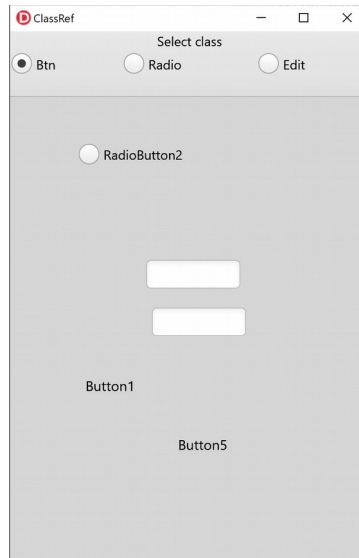
The first line of the code for this method is the key. It creates a new object of the class data type stored in the `FControlType` field. We accomplish this simply by applying the `Create` constructor to the class reference. Now you can set the value of the `Parent` property, set the position of the new component, give it a name (which is automatically used also as `Text`), and make it visible.

Notice in particular the code used to build the name; to mimic Object Pascal's default naming convention, I've taken the name of the class with the expression `FControlType.ClassName`, using a class method of the `TObject` class. Then I've added a number at the end of the name and removed the initial letter of the string.

356 - 12: Manipulating Classes

For the first radio button, the basic string is `TRadioButton`, plus the *1* at the end, and minus the *T* at the beginning of the class name—*RadioButton1*. Looks familiar?

Figure 12.1:
An example of the
output of the ClassRef
application, under
Window



You can see an example of the output of this program in Figure 12.1. Notice that the naming is not exactly the same as used by the IDE, which uses a separate counter for each type of control. This program uses a single counter for all of the components, instead. If you place a radio button, a push button, and an edit box in a form of the `ClassRef` application, their names will be *RadioButton1*, *Button2*, and *Edit3*, as shows in the image (although the edit has no visible description of its name).

As an aside, consider that once you've created a generic component, you can access its properties in a very dynamic way, using reflection, a topic covered in detail in Chapter 16. In that same chapter we'll see there are other ways to refer to type and class information beside class references.

Class And Record Helpers

As we have seen in Chapter 8, the concept of inheritance among classes is a way to expand a class providing new features, without affecting the original implementa-

tion in any way. This is an implementation of the co-called *open-closed principle*: The data type is fully defined (closed), but still modifiable (open).

While type inheritance is an extremely powerful mechanism, there are scenarios in which it is not ideal. The fact is that when you are working with existing and complex libraries, you might want to be able to augment a data type without inheriting a new one. This is particularly true when objects are created in some automatic way, and replacing their creation can be extremely complex.

A rather obvious scenario for Object Pascal developers is the use of components. If you want to add a method to a component class, to provide some new behavior to it, you can indeed use inheritance but that implies: create the new derived type, create a package to install it, replace all existing components in forms and other design surfaces with the new component type (an operation that affects both the form definition and the source code file).

The alternative approach is to use a class or record helper. These special purpose data types can extend an existing type with new methods. Even if they do have a few limitations, class helpers let you handle a scenario like the one I just outlined simply by adding new methods to an existing component, without any need to modify the actual component type.

note We actually already saw an alternative approach for extending a library class without fully replacing its references, by using inheritance and a same name class, the *interposer class* idiom. I covered that idiom in the final section of the last chapter. Class helpers offer a cleaner model, however they cannot be used to replace virtual methods or implement an extra interface, as I did in the application of the last chapter.

Class Helpers

A class helper is a way to add methods and properties to a class you have *no power* to modify (like a library class). Using a class helper to extend a class in your own code is really unusual, as in this case you should generally just go ahead and change the actual class.

What you cannot do in a class helper is add instance data, given the data should live in the actual objects and these are defined by their original class, or touch the virtual methods, again defined in the physical structure of the original class. In other words, a helper class can only add to or replace non-virtual methods of an existing class. This way you'll be able to apply the new method to an object of the original class, even if that class has no clue about the existence of the method.

358 - 12: Manipulating Classes

If this is not clear, and it probably isn't, let's look at an example (taken from the `ClassHelperDemo` application project – which is just a demo of what you shouldn't do, use class helpers to augment your own class):

```
type
  TMyObject = class
    protected
      Value: Integer;
      Text: string;
    public
      procedure Increase;
    end;

  TMyObjectHelper = class helper for TMyObject
    public
      procedure Show;
    end;
```

The preceding code declares a class and a helper for this class. This means that for an object of type `TMyObject`, you can call the methods of the original class as well as the methods of the class helper:

```
Obj := TMyObject.Create;
Obj.Text := 'foo';
Obj.Show;
```

The helper class methods becomes part of the class and can use `Self` just like any other method to refer to the current object (of the class it helps because class helpers are not instantiated), as this code demonstrates:

```
procedure TMyObjectHelper.Show;
begin
  Show (Text + ' ' + IntToStr (Value) + ' -- ' +
    ClassName + ' -- ' + ToString);
end;
```

Finally, notice that a helper class method can override the original method. In the code I've added a `Show` method both to the class and to the helper class, but only the helper class method gets called!

Of course, it makes very little sense to declare a class and an extension to the same class using the class helper syntax in the same unit or even in the same program. I did this in the demo only to make it easier to understand the technicality of this syntax. Class helpers should *not* be used as a general language construct for developing application architectures, but are mostly aimed at extending library classes you don't have the source code for or you don't want to change to avoid future conflicts.

There are a few more rules that apply to class helpers. Class helper methods:

- can have different access specifiers than the original method in the class
- can be class methods or instance methods, class variables and properties

- can be virtual methods, which can be overridden in a derived class (although I find this a bit awkward in practical terms)
- can introduce extra constructors
- can add nested constants to the type definition

The only feature they lack by design is instance data. Also note that class helpers are enabled as they become visible in the scope. You need to add a `uses` statement referring to the unit that declares the class helper to see its methods, not just include it once in the compilation process.

note For quite some time, there was an error in the Delphi compiler that ended up allowing class helpers to access private fields of the class they helped, regardless of the unit in which the class was declared. This “hack” basically broke object-oriented programming encapsulation rules. To enforce visibility semantics, class and record helpers in most recent versions of Object Pascal compilers (starting with Delphi 10.1 Berlin) cannot access private members of the classes or records that they extend. This has indeed caused existing code not to work any more, code that was leveraging this hack (which was never intended to be a language feature).

A Class Helper for a List Box

A practical use of class helpers is in providing extra methods for library classes. The reason being you don't want to change those classes directly (even if you have the source code, you don't really want to edit core library sources) or inherit from them (as this would force you to replace the components in the forms at design time).

As an example, consider this simple case: you want a simple way to get the text of the current selection of a list box. Instead of writing the classic code:

```
| ListBox1.Items [ListBox1.ItemIndex]
```

you can define a class helper as follows (taken from the `ControlHelper` project):

```
type
  TListboxHelper = class helper for TListBox
    function ItemIndexValue: string;
  end;

function TListboxHelper.ItemIndexValue: string;
begin
  Result := '';
  if ItemIndex >= 0 then
    Result := Items [ItemIndex];
end;
```

Now you can refer to the selected item of the list box as:

```
| Show (ListBox1.ItemIndexValue);
```

This is just a very simple case, but it shows the idea in very practical terms.

Class Helpers and Inheritance

The most significant limitation of helpers is that you can have only one helper for each class at a given time. If the compiler encounters two helper classes, the second will replace the first one. There is no way to chain class helpers, that is, have a class helper that further extends a class already extended with another class helper.

A partial solution to this issue comes from the fact that you can introduce a class helper for a class and add a further class helper for an inherited class... but you can't directly inherit a class helper from another class helper. I don't really encourage getting into complex class helper structures, because they can really turn your code into some very convoluted code.

An example would be the `TGUID` record, a Windows data structure you can actually use across platforms in Object Pascal, that has a helper adding a few common capabilities:

```
type
  TGUIDHelper = record helper for TGUID
    class function Create(const B: TBytes): TGUID; overload; static;
    class function Create(const S: string): TGUID; overload; static;
    // ... more Create overloads omitted
    class function NewGuid: TGUID; static;
    function ToByteArray: TBytes;
    function ToString: string;
  end;
```

You may have noticed that `TGUIDHelper` is a record helper rather than a class helper. Yes, records can have helpers just like classes can.

Adding a Controls Enumeration with a Class Helper

Any Delphi component in the libraries automatically defines an enumerator you can use to process each of the owned components, or child components. For example in a form method, you can enumerate the components owned by the form by writing:

```
for var AComp in self do
  ... // use AComp
```

Another common operation is to navigate the child controls, which include only visual components (excluding non-visual components like a `TMainMenu`) that have the form as direct parent (excluding controls hosted by a child control, like a button on a panel). A technique we can use to write simpler code for cycling over child con-

trols is to add a new enumeration to the `TwinControl` class by writing the following class helper:

```
type
  TControlsEnumHelper = class helper for TwinControl
  type
    TControlsEnum = class
      private
        NPos: Integer;
        FControl: TwinControl;
      public
        constructor Create (aControl: TwinControl);
        function MoveNext: Boolean;
        function GetCurrent: TControl;
        property Current: TControl read GetCurrent;
      end;
    public
      function GetEnumerator: TControlsEnum;
    end;
```

note The reason the helper is for `TwinControl` and not `TControl` is that only controls with a windows handle can be the parent of other controls. This basically exclude graphical controls.

This is the complete code of the helper, including its only method and those of the nested class `TControlsEnum`:

```
{ TControlsEnumHelper }

function TControlsEnumHelper.GetEnumerator: TControlsEnum;
begin
  Result := TControlsEnum.Create (self);
end;

{ TControlsEnumHelper.TControlsEnum }

constructor TcontrolsEnumHelper.TcontrolsEnum.Create(
  aControl: TwinControl);
begin
  FControl := aControl;
  NPos := -1;
end;

function TControlsEnumHelper.TControlsEnum.GetCurrent: TControl;
begin
  Result := FControl.Controls[NPos];
end;

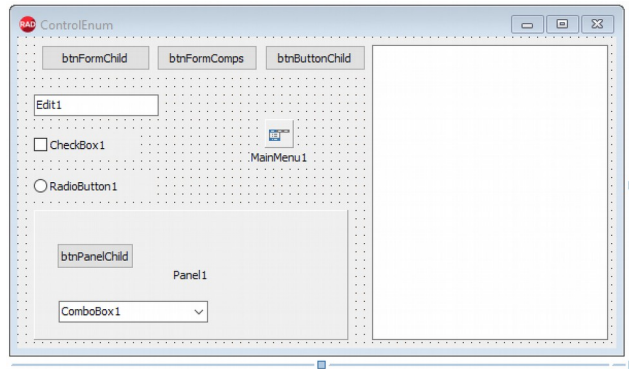
function TControlsEnumHelper.TControlsEnum.MoveNext: Boolean;
begin
  Inc (NPos);
  Result := NPos < FControl.ControlCount;
end;
```

362 - 12: Manipulating Classes

Now if we create a form like that in Figure 12.2, we can test the enumeration in various scenarios. The first case is what we specifically wrote the code for, that is enumerate the child controls of the form:

Figure 12.2:

The form used for testing the controls enumeration helper at design time in the Delphi IDE



```
procedure TControlEnumForm.BtnFormChildClick(Sender: TObject);  
begin  
    Memo1.Lines.Add ('Form Child');  
    for var ACtrl in self do  
        Memo1.Lines.Add (ACtrl.Name);  
    Memo1.Lines.Add ('');  
end;
```

This is the output of the operation in the memo control, listing the form child controls, but not other components or controls parented by the panel:

```
Form Child  
Memo1  
BtnFormChild  
Edit1  
CheckBox1  
RadioButton1  
Panel1  
BtnFormComps  
BtnButtonChild
```

The complete list will show up if we enumerate all component. However, we have an issue using the code I showed at the beginning of this section, because we have overridden the `GetNumerator` method with a new version (in the class helper) and for this reason we cannot directly access to the base `TComponent` enumerator. The helper is defined for `TWinControl`, so we can use a trick. If we cast our object to `TComponent` the code will invoke the standard, predefined enumerator:

```
procedure TControlEnumForm.BtnFormCompsClick(Sender: TObject);
```

```
begin
  Memo1.Lines.Add ('Form Components');
  for var AComp in TComponent(self) do
    Memo1.Lines.Add (AComp.Name);
  Memo1.Lines.Add ('');
end;
```

This is the output, listing more components than the previous list:

```
Form Components
Memo1
BtnFormChild
Edit1
CheckBox1
RadioButton1
Panel1
BtnPanelChild
ComboBox1
BtnFormComps
BtnButtonChild
MainMenu1
```

In the ControlsEnum application project I've also added code for enumerating the child controls of the panel and those of one of the buttons (mostly to test that the enumerator works correctly when the list is empty).

Record Helpers for Intrinsic Types

A further extension of the record helper concepts is the ability to add methods to native (or *compiler intrinsic*) data types. Although the same “record helper” syntax is used, this is not applied to records but to regular data types.

note Record helpers are currently used to augment and add method-like operations to native data types, but this might as well change in the future. Today's run-time library defines a few native helpers that might disappear in the future, preserving the way you write code that uses those helpers... but breaking compatibility in the code that defines them. That's why you should not overuse this feature, even if it certainly very nice and handy.

How do intrinsic type helpers work in practice? Let's consider the following definition of a helper for the `Integer` type:

```
type
  TIntHelper = record helper for Integer
    function AsString: string;
  end;
```

Now given an `Integer` variable `N`, you can write:

```
N.AsString;
```

364 - 12: Manipulating Classes

How do you define that pseudo-method and how can it refer to the variable's value? By stretching the meaning of the `self` keyword to refer to the value the function is applied to:

```
function TIntHelper.AsString: string;  
begin  
    Result := IntToStr (self);  
end;
```

Notice that you can apply methods also to constants, like in:

```
    Caption := 400000.AsString;
```

However, you cannot do the same for a small value, as the compiler interprets constants of the smaller possible type. So if you want to get out the value 4 as a string you have to use the second form:

```
    Caption := 4.AsString; // nope!  
    Caption := Integer(4).AsString; // ok
```

Or you can make the first statement to compile by defining a different helper:

```
type  
    TByteHelper = record helper for Byte...
```

As we already saw in Chapter 2, you don't really need to write the code above for types like `Integer` and `Byte`, as the run-time library defines a pretty comprehensive list of class helpers for most core data types, including the following which are defined in the `System.SysUtils` unit:

```
TStringHelper = record helper for string  
TSingleHelper = record helper for Single  
TDoubleHelper = record helper for Double  
TExtendedHelper = record helper for Extended  
TByteHelper = record helper for Byte  
TShortIntHelper = record helper for ShortInt  
TSmallIntHelper = record helper for SmallInt  
TWordHelper = record helper for word  
TCardinalHelper = record helper for Cardinal  
TIntegerHelper = record helper for Integer  
TInt64Helper = record helper for Int64  
TUInt64Helper = record helper for UInt64  
TNativeIntHelper = record helper for NativeInt  
TNativeUIntHelper = record helper for NativeUInt  
TBooleanHelper = record helper for Boolean  
TByteBoolHelper = record helper for ByteBool  
TWordBoolHelper = record helper for WordBool  
TLongBoolHelper = record helper for LongBool  
TWordBoolHelper = record helper for WordBool
```

There are a few other intrinsic type helpers currently defined in other units, like:

```
// System.Character:  
TCharHelper = record helper for Char  
// System.Classes:
```

```

| TUInt32Helper = record helper for UInt32

```

Given I've covered the use of these helpers in many examples in the initial part of the book, there is no need to reiterate them here. What this section added is a description of how you can define an intrinsic type helper.

Helpers for Type Aliases

As we saw, it isn't possible to define two helpers for the same type, let alone an intrinsic type. So how do you add an extra direct operation to a native type, like Integer? While there is no clear cut solution, there are some possible workarounds (short of copying the internal class helper source code and duplicating it with the extra method).

A solution I like is the definition of a type alias. A type alias is seen as a brand new type by the compiler, so it can have its own helper without replacing the helper of the original type. Now given the types are separate, you still cannot apply methods of both class helpers to the same variable, but one of the sets will be a type cast away. Let me explain this in code terms. Suppose you create a type alias like:

```

| type
  MyInt = type Integer;

```

Now you can define a helper for this type:

```

| type
  TMyIntHelper = record helper for MyInt
    function AsString: string;
  end;

function MyIntHelper.AsString: string;
begin
  Result := IntToStr (self);
end;

```

If you declare a variable of this new type, you can invoke the methods specific helper, but still call the Integer type helper methods with a cast:

```

| procedure TForm1.Button1Click(Sender: TObject);
var
  mi: MyInt;
begin
  mi := 10;
  Show (mi.AsString);
  // Show (mi.ToString); // this doesn't work
  Show (Integer(mi).ToString)
end;

```

366 - 12: Manipulating Classes

This code is in the `TypeAliasHelper` application project, for you to try out further variations.

13: objects and memory

This chapter focused on a very specific and quite important topic, that is memory management in the Object Pascal language. The language and its run-time environment offer a rather unique solution, which is in-between C++ style manual memory management and Java or C# garbage collection.

The reason for this in-between approach is it helps avoiding most of the manual memory managements hassles (but clearly not all), without the constraints and the problems caused by garbage collection, from extra memory allocation to non-deterministic disposal.

note I have no particular intention of delving into the problems of GC (Garbage Collection) strategies and how they are implemented in the various platforms. This is more of a research topic. What is relevant is that on constrained devices like mobile ones, GC seems to be far from ideal, but some of the same issues apply on every platform. The trend of ignoring the memory consumption of Windows applications has brought us small utilities with take 1 GB each.

What makes things a little extra complicated in Object Pascal, though, is that fact that a variable uses memory depending on its data type, with some types using ref-

368 - 13: Objects and Memory

erence counting and some a more traditional approach. The component-based ownership model and a few other options make memory management a complex topic. This chapter is here to address it, starting with some of the foundations of the memory management in modern programming languages and the concepts behind the object reference model.

note For many years, Delphi mobile compilers offered a different memory model called ARC, or Automatic Reference Counting. Promoted by Apple in its own languages, ARC adds compiler support for tracking and counting references to an object, destroying it when it is no longer needed (that is, the reference count goes down to zero). This is very similar to what happens with interface references in Delphi on all platforms. Starting with Delphi 10.4 the support for ARC was removed from the language for all platforms.

Global Data, Stack, and Heap

The memory used by any Object Pascal application on any platform can be divided into two areas: code and data. Portions of the executable file of a program, of its resources (bitmaps and form description files), and of the libraries used by the program are loaded in its memory space. These memory blocks are read-only, and (on some platforms like Windows) they can be shared among multiple processes.

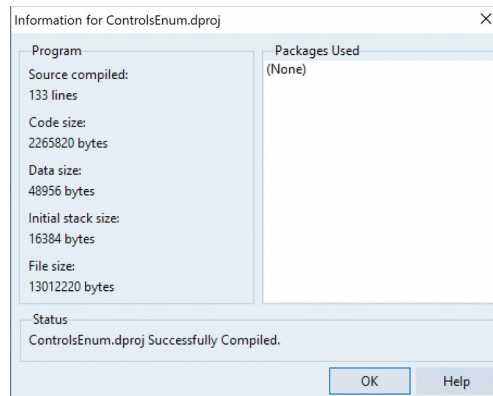
It is more interesting to look at the data portion. The data of an Object Pascal program (like that of programs written in most other languages) is stored in three clearly distinct areas: the global memory, the stack, and the heap.

Global Memory

When the Object Pascal compiler generates the executable file, it determines the space required to store variables that exist for the entire lifetime of the program. Global variables declared in the interface or in the implementation portions of a unit fall into this category. Note that if a global variable is of a class type (but also a string or a dynamic array), only a 4-byte or 8-byte object reference is stored in the global memory.

You can determine the size of the global memory by using the Project | Information menu item after compiling a program. The specific field you want to look at is *Data size*. Figure 13.1, shows a usage of almost 50K of global data (48,956 bytes), which includes global data of both your program and the libraries you use.

Figure 13.1: The information about a compiled program



Global memory is sometimes called static memory as once your program is loaded the variables remain in their original location or the memory is never released.

Stack

The *stack* is a dynamic memory area, which is allocated and deallocated following the LIFO order: Last In, First Out. This means that the last memory object you've allocated will be the first to be deleted. You can see a representation of the stack memory in Figure 13.2.

Stack memory is typically used by routines (procedure, function, and method calls) for passing parameters and their return values and for the local variables you declare within a function or method. Once a routine call is terminated, its memory area on the stack is released. Remember, anyway, that using Object Pascal's default register-calling convention, the parameters are passed in CPU registers instead of the stack whenever possible.

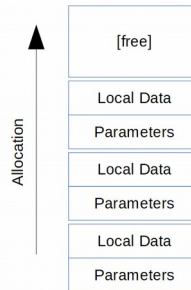
Notice also that stack memory is generally not initialized nor cleaned up, to save time. This is why if you declare, say, an Integer as a local variable and just read its value, you can find pretty much everything. All local variables need to be initialized before they are used.

The size of the stack is generally fixed and determined by the compilation process. You can set this parameter in the linker page of the Project options. However, the default is generally OK. If you receive a "stack overflow" error message, this is probably because you have a function recursively calling itself forever, not because the

370 - 13: Objects and Memory

stack space is too limited. The initial stack size is another piece of information provided by the Project | Information dialog.

Figure 13.2: A representation of the stack memory area



Heap

The *heap* is the area in which the allocation and deallocation of memory happens in random order. This means that if you allocate three blocks of memory in sequence, they can be destroyed later on in any order. The heap manager takes care of all the details, so you simply ask for new memory with the low-level `GetMem` function or by calling a constructor to create an object, and the system will return a new memory block for you (possibly reusing memory blocks already discarded). Object Pascal uses the heap for allocating the memory of each and every object, the text of strings, for dynamic arrays, and for most other data structures.

Because it is dynamic, the heap is the memory area where programs generally have the most problems:

- Every time an object is created, it needs to be destroyed. Failing to do so is a scenario called “memory leak”, which won't do too much harm unless it is repeated over and over until the entire heap memory is full.
- Every time an object is destroyed, you have to make sure it is not used any more, and that the program doesn't try to destroy it a second time.

The same is true for any other dynamically created data structure, but the language run-time looks after strings and dynamic arrays in a mostly automatic way, so you almost never have to worry about those.

The Object Reference Model

As we have seen in Chapter 7, objects in the language are implemented as references. A variable of a class type is just a pointer to the memory location on the heap where the object data lives. There is actually a little extra information, like a class reference, a way to access the object virtual methods table, but this is outside of the focus of this chapter (I'll shortly introduce it in the section “Is this Pointer an Object Reference?” of Chapter 13).

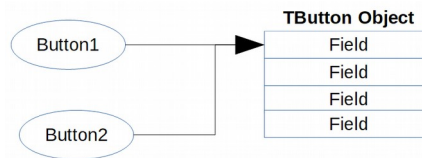
We have also seen how assigning an object to another only makes a copy of the reference, so you'll have two references to a single object in memory. To have two completely separate objects, you need to create a second one and copy the first object's data to it (an operation not available automatically, as its implementation details could vary depending on the actual data structure).

In coding terms, if you write the following code you don't create a new object, but rather a new reference to the same object in memory:

```
var
  Button2: TButton;
begin
  Button2 := Button1;
```

In other words, there is only one object in memory, and both the `Button1` and `Button2` variables refer to it, as you can see in Figure 13.3.

Figure 13.3: Copying object references



Passing Objects as Parameters

Something similar happens when you are passing an object as a parameter to a function or a method. In general terms, you are just copying the reference to the same object, and within the method or function you can do operations on that object and modify it, regardless of the fact the parameter is passed as a `const` parameter.

For example, by writing this procedure and calling it as follows, you'll modify the caption of `Button1`, or `AButton` if you prefer:

372 - 13: Objects and Memory

```
procedure ChangeCaption (AButton: TButton; Text: string);  
begin  
    AButton.Text := Text;  
end;  
  
// call...  
ChangeCaption (Button1, 'Hello')
```

What if you need to create a new object, instead? You'll basically have to create it and then copy each of the relevant properties. Some classes, notably most classes derived from `TPersistent`, and not from `TComponent`, define an `Assign` method to copy the data of an object. For example, you can write

```
ListBox1.Items.Assign (Memo1.Lines);
```

Even if you assign those properties directly, Object Pascal will execute similar code for you. In fact, the `SetItems` method connected with the `items` property of the list box calls the `Assign` method of the `TStringList` class representing the actual items of the list box.

So let's try to recap what the various parameters passing modifiers do when applied to objects:

- If there is a **no modifier**, you can do any operation on the object and the variable referring to it. You can modify the original object, but if you assign a new object to the parameter this new object will have nothing to do with the original one and the variable referring to it.
- If there is a **const modifier**, you can change values and call methods of the object, but you cannot assign a new object to the parameter. Notice that there is no performance advantage in passing an object as `const`.
- If there is a **var modifier**, you can change anything in the object and also replace the original object with a new one in the calling location, as happens with other `var` parameters. The constraint is that you need to pass a reference to a variable (not a general expression) and that the reference type must match the parameter type exactly.
- Finally there is a fairly unknown option for passing an object as parameter, called **constant reference** and written as `[ref] const`. When a parameter is passed as constant reference it behaves similarly to a passing by reference (`var`), but it allows more flexibility in the type of the parameter you are passing, not requiring an exact type match (as passing an object of a subclass is allowed).

Memory Management Tips

Memory management in Object Pascal is subject to two simple rules: You must destroy every object and memory block you create and allocate, and you must destroy each object and free each block only once. Object Pascal supports three types of memory management for dynamic elements (that is, elements not in the stack and the global memory area), detailed in this remaining part of this section:

- Every time you create an object, you should also free it. If you fail to do so, the memory used by that object won't be released for other objects, until the program terminates.
- When you create a component, you can specify an owner component, passing the owner to the component constructor. The owner component (often a form or data module) becomes responsible for destroying all the objects it owns. In other words, when you free the form or data module, it frees all of the components it owns. So, if you create a component and give it an owner, you don't have to worry about destroying it.
- When you allocate memory for strings, dynamic arrays, and objects referenced by interface variables (as discussed Chapter 11), Object Pascal automatically frees the memory when the reference goes out of scope. You don't need to free a string: when it becomes unreachable, its memory is released.

Destroying Objects You Create

In the most simple scenario, on desktop compilers you have to create the temporary objects you destroy. Any non-temporary object should have an owner, be part of a collection, or be reachable through some data structure, which will become responsible for destroying it in due time.

The code used to create and destroy a temporary object is generally encapsulated in a `try-finally` block, so that the object is destroyed even if something goes wrong when using it:

```
MyObj := TMyClass.Create;
try
  MyObj.DoSomething;
finally
  MyObj.Free;
end;
```

374 - 13: Objects and Memory

Another common scenario is that an object is used by another one, which becomes its owner:

```
constructor TMyOwner.Create;  
begin  
    FSubObj := TSubObject.Create;  
end;  
  
destructor TMyOwner.Destroy;  
begin  
    FSubObj.Free;  
end;
```

There are some common more complex scenarios, in case the subject is not created until needed (lazy initialization) or might be destroyed before the owner, in case it is not needed any more. To implement lazy initialization, you don't create the subject in the owner object constructor, but when it is needed:

```
function TMyOwner.GetSubObject: TSubObject  
begin  
    if not Assigned (FSubObj) then  
        FSubObj := TSubObject.Create;  
    Result := FSubObj;  
end;  
  
destructor TMyOwner.Destroy;  
begin  
    FSubObj.Free;  
end;
```

Notice you don't need to test if the object is assigned before freeing it, because this is exactly what `Free` does, as we'll see in the next section.

Destroying Objects Only Once

Another problem is that if you call the destructor of an object twice, you get an error. A *destructor* is a method that deallocates an object's memory. We can write code for a destructor, generally overriding the default `Destroy` destructor, to let the object execute some code before it is destroyed.

`Destroy` is a virtual destructor of the `TObject` class. Most of the classes that require custom clean-up code when the objects are destroyed override this virtual method. The reason you should never define a new destructor is that objects are usually destroyed by calling the `Free` method, and this method calls the `Destroy` virtual destructor (possibly the overridden version) for you.

As I've just mentioned, `Free` is simply a method of the `TObject` class, inherited by all other classes. The `Free` method basically checks whether the current object (`Self`) is not `nil` before calling the `Destroy` virtual destructor.

note You might wonder why you can safely call `Free` if the object reference is `nil`, but you can't call `Destroy`. The reason is that `Free` is a known method at a given memory location, whereas the virtual function `Destroy` is determined at run time by looking at the type of the object, a very dangerous operation if the object doesn't exist any more.

Here is its pseudo-code for `Free`:

```
procedure TObject.Free;
begin
  if Self <> nil then
    Destroy;
end;
```

Next, we can turn our attention to the `Assigned` function. When we pass a pointer to this function, it simply tests whether the pointer is `nil`. So the following two statements are equivalent, at least in most cases:

```
if Assigned (MyObj) then
  ...
if MyObj <> nil then
  ...
```

Notice that these statements test only whether the pointer is not `nil`; they do not check whether it is a valid pointer. If you write the following code

```
MyObj.Free;
if MyObj <> nil then
  MyObj.DoSomething;
```

the test will evaluate to `True`, and you'll get an error on the line with the call to the method of the object. It is important to realize that calling `Free` doesn't set an object's reference to `nil`.

Automatically setting an object to `nil` is not possible. You might have several references to the same object, and Object Pascal doesn't track them. At the same time, within a method (such as `Free`) we can operate on the object, but we know nothing about the object reference—the memory address of the variable we've used to call the method.

In other words, inside the `Free` method or any other method of a class, we know the memory address of the object (`Self`), but we don't know the memory location of the variable referring to the object, such as `MyObj`. Therefore, the `Free` method cannot affect the `MyObj` variable.

However, when we call an external function passing an object as parameter by reference, which allows this function to modify the original value of the parameter. The

376 - 13: Objects and Memory

ready-to-use function for this scope if the `FreeAndNil` procedure. Here is current code for `FreeAndNil`:

```
procedure FreeAndNil(const [ref] Obj: TObject); inline;  
var  
    Temp: TObject;  
begin  
    Temp := Obj;  
    TObject(Pointer(@Obj)^) := nil;  
    Temp.Free;  
end;
```

In the past, the parameter was just a pointer, but the drawback was you could pass to the `FreeAndNil` procedure also a raw pointer, an interface references and other incompatible data structures. This would often cause memory corruption and hard-to-find bugs. Starting with Delphi 10.4 the code was modified as shown above, using a `const` reference parameter of the `TObject` type, limiting the parameters to objects.

note Quite a few Delphi experts would argue that `FreeAndNil` should never be used, because the visibility of the variable referring to an object should match its lifetime. If an object owns another and frees it in the destructor, there is no need to set the reference to `nil` as it is part of an object you are no longer going to use. Similarly, a local variable with a `try finally` block freeing it, doesn't need to set it to `nil` as it is about to go out of scope.

As a side note, beside the `Free` method, `TObject` has also a `DisposeOf` method which is a left over of the ARC support the language had for a few years. Currently, the `DisposeOf` method just calls `Free`.

To sum things up on the use of these memory cleanup operations, here are a couple of guidelines:

- Always call `Free` to destroy objects, instead of calling the `Destroy` destructor.
- Use `FreeAndNil`, or set object references to `nil` after calling `Free`, unless the reference is going out of scope immediately afterward.

Memory Management and Interfaces

In Chapter 11 I introduced the key elements of memory management for interfaces, which unlike objects are managed and reference counted. As I mentioned, interface references increase the reference count of the referenced object, but you can declare an interface reference as weak to disable the reference counting (but still ask the compiler to manage the reference for you) or you can use the `unsafe` modifier to

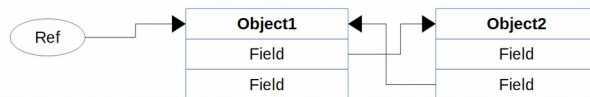
completely disable any compiler support for the specific reference. In this section we'll go a bit deeper in this area, showing some additional example to what was provided in Chapter 11.

More on Weak References

One issue with the reference counting models Delphi uses for interfaces, is that if two objects refer to one another they form a circular reference, and their reference count will basically never get to zero. Weak references offer a mechanism to *break* these cycles, allowing you to define a reference that doesn't increase the reference count.

Suppose that two interfaces refer to each other using one of their fields, and an external variable refers to the first. The reference count of the first object will be 2 (the external variable, and the field of the second object): while the reference count of the second object is 1 (the field of the first object). Figure 13.4 depicts this scenario.

Figure 13.4:
References among
objects can form cycles,
something weak
references account for.



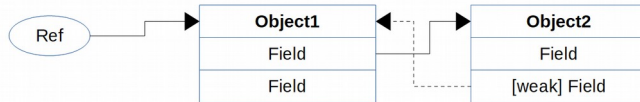
Now, as the external variable goes out of scope, the two objects' reference counts remain 1, and they'll remain in memory indefinitely. To solve this type of situation, you should break the circular references, something far from simple, given that you don't know when to perform this operation (it should be performed when the last external reference goes out of scope, a fact of which the objects have no knowledge). The solution to this situation, and many similar scenarios, is to use a weak reference. As mentioned, a weak reference is a reference to an object that doesn't increase its reference count. Technically, you define a weak reference by applying the `[weak]` attribute to it.

note Attributes are an advanced Object Pascal language feature covered in Chapter 16. Suffice to say that they are a way to add some run-time information about a symbol, so that external code can determine how to handle it.

378 - 13: Objects and Memory

Given the previous scenario, if the reference from the second object back to the first one is a weak reference (see Figure 13.5), as the external variable goes out of scope, both objects will be destroyed.

Figure 13.5: The references cycle in Figure 13.4, broken using a weak reference (dotted line).



Let's look at this simple situation in code. First of all, the ArcExperiments application sample declares two interfaces, with one referring to the other:

```
type
  MySimpleInterface = interface
    ['{B6AB548A-55A1-4D0F-A2C5-726733C33708}']
    procedure DoSomething(bRaise: Boolean = False);
    function RefCount: Integer;
  end;

  MyComplexInterface = interface
    ['{5E8F7B29-3270-44FC-B0FC-A69498DA4C20}']
    function GetSimple: MySimpleInterface;
    function RefCount: Integer;
  end;
```

The code of the program defines two different classes, which implement each one of the interfaces. Notice how the cross references (FOwnedBy and FSimple are based on interfaces and one of the two is defined as weak):

```
type
  TMySimpleClass = class (TInterfacedObject, MySimpleInterface)
  private
    [weak] FOwnedBy: MyComplexInterface;
  public
    constructor Create(Owner: MyComplexInterface);
    destructor Destroy (); override;
    procedure DoSomething(bRaise: Boolean = False);
    function RefCount: Integer;
  end;

  TMyComplexClass = class (TInterfacedObject, MyComplexInterface)
  private
    FSimple: MySimpleInterface;
  public
    constructor Create();
    destructor Destroy (); override;
    function GetSimple: MySimpleInterface;
    function RefCount: Integer;
  end;
```

Here the constructor of the “complex” class creates an object of the other class:

```
constructor TMyComplexClass.Create;
begin
  inherited Create;
  FSimple := TMySimpleClass.Create (self);
end;
```

Remember that the `FOwnedBy` field is a weak reference, so it doesn't increase the reference count of the object it refers to, in this case the current object (`self`). Given this code structure, we can write:

```
class procedure TMyComplexClass.CreateOnly;
var
  MyComplex: IMyComplexInterface;
begin
  MyComplex := TMyComplexClass.Create;
  MyComplex.FSimple.DoSomething;
end;
```

This will cause no memory leak, as long as the weak reference is used. For example, with code like:

```
var
  MyComplex: IMyComplexInterface;
begin
  MyComplex := TMyComplexClass.Create;
  Log ('Complex = ' + MyComplex.RefCount.ToString);
  MyComplex.GetSimple.DoSomething (False);
```

Given each constructor and destructor logs its execution, you'll get a log like:

```
Complex class created
Simple class created
Complex = 1
Simple class doing something
Complex class destroyed
Simple class destroyed
```

If you remove the weak attribute in the code, you'll see a memory leak and also (in the execution of the code above) a value for the reference count of 2 rather than 1.

Weak References Are Managed

A very important element is that weak references are managed. In other words, the system keeps a list of the weak references in memory, and when an object is destroyed it checks if there is a weak reference referring to it, and it sets it to `nil`. This means weak references have a run-time cost.

The good thing about having managed weak references, compared to traditional ones, is that you can check if an interface reference is still valid or not (meaning the

380 - 13: Objects and Memory

object it is referring to has been destroyed). When you are using a weak reference, you should always test if it is assigned before using it.

In the `ArcExperiments` application sample, the form has a private field of the `IMySimpleInterface` type, declared as a weak reference:

```
private
[weak] MySimple: IMySimpleInterface;
```

There is also a button assigning a reference to that field, and a different one using it, after making sure it is still valid:

```
procedure TForm3.BtnGetWeakClick(Sender: TObject);
var
  MyComplex: IMyComplexInterface;
begin
  MyComplex := TMyComplexClass.Create;
  MyComplex.GetSimple.DoSomething (False);
  MySimple := MyComplex.GetSimple;
end;

procedure TForm3.BtnUseWeakClick(Sender: TObject);
begin
  if Assigned (MySimple) then
    MySimple.DoSomething(False)
  else
    Log ('Nil weak reference');
end;
```

Unless you modify the code, that `if Assigned` test will fail because the first button event handler creates and immediately releases the objects, so that the weak reference becomes *invalid*. But given it is managed, the compiler helps you track its real status (unlike a reference to an object).

The Unsafe Attribute

There are some very specific circumstances (for example, during the creation of an instance) in which a function might return an object with a reference count set to zero. In this case, in order to avoid the compiler deleting the object right away (before it has a chance to be assigned to a variable, which would increase its reference count to 1), we have to mark the object as “*unsafe*”.

The implication here is that its reference count has to be temporarily ignored, to make the code “safe”. This behavior is accomplished by using a new specific attribute, `[Unsafe]`, a feature you should need only in very specific circumstances. This is the syntax:

```
var
[Unsafe] Intf1: IInterface;
```

```
[Result: Unsafe] function GetIntf: IInterface;
```

The use of this attribute can make sense when implementing construction patterns, like the factory pattern, in general purpose libraries.

note To support now deprecated ARC memory model, the System unit used an `unsafe` directive, because it couldn't use the attribute before its definition (later in the same unit). This isn't supposed to be used in any code outside of that unit, and it is not used any more (you can see it within a `$IFDEF` directive).

Tracking and Checking Memory

We have seen in this chapter the foundations of memory management in Object Pascal. In most cases, just applying the rules highlighted here will suffice to keep your programs stable, avoid excessive memory usage, and basically let you forget about memory management. There are some further best practices for writing robust applications covered later in this chapter.

In this section I'm focusing on the techniques you can use to track memory usage, monitor anomalous situations, and find memory leaks. This is important knowledge for a developer, even if it isn't strictly part of the language, but more of the run-time support. Also, the implementation of the memory manager depends on the target platform and operating system, and you can even plug-in a custom memory manager in an Object Pascal application (not that this is a common though).

Notice that all of the discussion related with tracking memory status, memory managers, leak detection relate only to *heap memory*. The stack and the global memory are managed differently and you basically have no power to intervene, but these are also memory areas that rarely cause any trouble.

Memory Status

What about tracking heap memory status? The RTL offers a couple of handy functions, `GetMemoryManagerState` and `GetMemoryMap`. While the memory manager state is an indication of the number of allocated blocks of various sizes, the heap map is quite nice as it depicts the memory status of the application at the system level. You can examine the actual status of each following memory block by writing code like:

382 - 13: Objects and Memory

```
for I := Low(aMemoryMap) to High(aMemoryMap) do
begin
  case AMemoryMap[I] of
    csUnallocated: ...
    csAllocated: ...
    csReserved: ...
    csSysAllocated: ...
    csSysReserved: ...
  end;
end;
```

This code is used in the `ShowMemory` application project to create a graphical representation of the application memory status.

FastMM4

On the Windows platform, the current Object Pascal memory manager is called FastMM4 and was developed as an open source project mainly by Pierre La Riche. FastMM4 optimizes the allocation of memory, speeding it up and freeing more RAM for subsequent use. FastMM4 is capable of doing extensive memory checks on effective memory clean-up, on incorrect use of deleted objects, including interface based access for that data, on memory overwrites and on buffer overruns. It can also provide some feedback on left-over objects, helping you track memory leaks.

Actually some of the more advanced features of FastMM4 are available only in the full version of the library (covered in the section “Buffer Overruns in the Full FastMM4”), not in the version that is included in the standard RTL. This is why if you want to have the capabilities of the full version, you have to download its full source code from:

■ <https://github.com/pleriche/FastMM4>

note There is a new version of this library called FastMM5, which has been specifically optimized for multithreaded applications and can perform much better on large multi-core systems. The new version of the library is available with a GPL license (for open source projects) or with a paid commercial license (fully worth the cost) for anyone else. More information is available in the *read me* of the project at <https://github.com/pleriche/FastMM5>.

Tracking Leaks and Other Global Settings

The RTL version of the FastMM4 can be tuned using global settings in the `System` unit. Notice that while the relevant global declarations are in the `System` unit, the actual memory manager is implemented in the `getmem.inc` RTL source code file.

Again, this is active by default only for Windows applications, not for other operating systems, that use their platform native memory manager.

The easiest-to-use setting is the `ReportMemoryLeaksOnShutdown` global variable, which allows you to easily track a memory leak. You need to turn it on at the beginning of the program execution, and when the program terminates it will tell you if there are any memory leaks in your code (or in any of the libraries you are using).

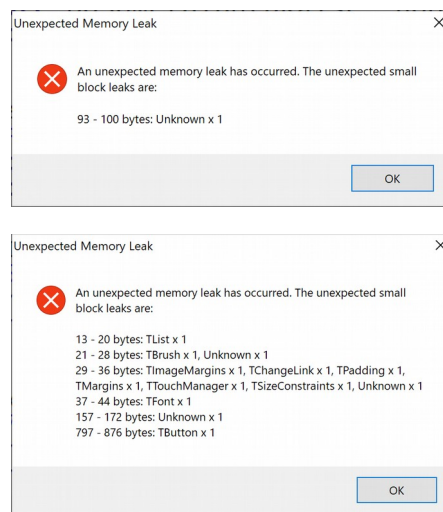
note More advanced settings of the memory manager include the `NeverSleepOnMMThreadContention` global variable for multithreaded allocations; the functions `GetMinimumBlockAlignment` and `SetMinimumBlockAlignment`, which can speed up some SSE operation to the expense of using more memory; the ability to register an expected memory leak by calling the global procedure `RegisterExpectedMemoryLeak`.

To demonstrate standard memory leak reporting and registration, I've written the simple `LeakTest` application project. It has a button with this `onClick` handler:

```
var
  P: Pointer;
begin
  GetMem (P, 100); // leak!
end;
```

This code allocates 100 bytes that are lost... or leaked. If you run the `LeakTest` program while the IDE is running and press the first button once, as you close the program you'll get a message like the one in the upper part of Figure 13.6.

Figure 13.6: The memory leaks reported by the memory manager on Windows upon termination of the `LeakTest` application



384 - 13: Objects and Memory

The other “leak” of the program is caused by creating and leaving in memory a `TButton`, but as this object includes many sub-elements, the leak report becomes more complex, like the one in the bottom part of Figure 13.6. Still, we do have some limited information about the leak itself.

The program also allocates some memory for a global pointer that's never going to be freed, but by registering this potential leak as expected, it won't get reported:

```
procedure TFormLeakTest.FormCreate(Sender: TObject);  
begin  
    GetMem (GlobalPointer, 200);  
    RegisterExpectedMemoryLeak(GlobalPointer);  
end;
```

Again, this basic leak reporting is only available by default only on the Windows platform, where FastMM4 is actually used by default.

Buffer Overruns in the Full FastMM4

This is a rather advanced topic, and one that is specific to the Windows platform, so I'd recommend only the most experienced developers read this section.

If you want to have more control over leak reporting (like activating file-based logging), fine tune the allocation strategy and use memory checks provided by FastMM4, you need to download the full version. This consists of the `FastMM4.pas` file plus the configuration file `FastMM4Options.inc`.

It is the latter file that you need to edit to fine tune the settings, simply by commenting and uncommenting a large number of directives. By convention, this is done by placing a period before the `$DEFINE` statement, turning it into a plain comment, as in the first of these two lines taken from the include file:

```
{.$DEFINE Align16Bytes} // comment  
{.$DEFINE UseCustomFixedSizeMoveRoutines} // active setting
```

For this demo I've turned on the following relevant settings, reported here to give you an idea of the kind of available defines:

```
{.$DEFINE FullDebugMode}  
{.$DEFINE LogErrorsToFile}  
{.$DEFINE EnableMemoryLeakReporting}  
{.$DEFINE HideExpectedLeaksRegisteredByPointer}  
{.$DEFINE RequireDebuggerPresenceForLeakReporting}
```

The test program (in the folder `FastMMCode`, which also includes the full source of the version of FastMM4 that I used, for your convenience) activates the custom ver-

sion of the memory manager in the project source code file, by setting it as the first unit:

```
program FastMMCode;

uses
  FastMM4 in 'FastMM4.pas',
  Forms,
  FastMMForm in 'FastMMForm.pas'; {Form1}
```

You'll also need a local copy of the `FastMM_FullDebugMode.dll` file to make it work. This demo program causes a buffer overrun by getting more text than it can fit in the local buffer, as `Length(Caption)` is larger than the 5 characters provided:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  pch1: PChar;
begin
  GetMem (pch1, 5);
  GetWindowText(Handle, pch1, Length(Caption));
  ShowMessage (pch1);
  FreeMem (pch1);
end;
```

The memory manager allocates extra bytes at the beginning and at the end of each memory block with special values, and checks for those values when you free each memory block. This is why you get the error on the `FreeMem` call. As you press the button (in the debugger) you'll see a very long error message, which is also logged to the file:

```
FastMMCode_MemoryManager_EventLog.txt
```

This is the output of the overrun error, with stack traces at the time of the allocation and free operations, plus the current stack trace and a memory dump (partial here):

```
FastMM has detected an error during a FreeMem operation. The block
footer has been corrupted.
```

```
The block size is: 5
```

```
Stack trace of when this block was allocated (return addresses):
```

```
40305E [System][System.@GetMem]
44091A [Controls][Controls.TControl.Click]
44431B [Controls][Controls.TWinControl.WndProc]
42D959 [StdCtrls][StdCtrls.TButtonControl.WndProc]
44446C [Controls][Controls.DoControlMsg]
44431B [Controls][Controls.TWinControl.WndProc]
45498A [Forms][Forms.TCustomForm.WndProc]
443A43 [Controls][Controls.TWinControl.MainWndProc]
41F31A [Classes][Classes.StdWndProc]
76281A10 [GetMessageW]
```

```
The block is currently used for an object of class: Unknown
```

386 - 13: Objects and Memory

The allocation number is: 381

Stack trace of when the block was previously freed (return addresses):

```
40307A [System][System.@FreeMem]
42DB8A [StdCtrls][StdCtrls.TButton.CreateWnd]
443863 [Controls][Controls.TwinControl.UpdateShowing]
44392B [Controls][Controls.TwinControl.UpdateControlState]
44431B [Controls][Controls.TwinControl.WndProc]
45498A [Forms][Forms.TCustomForm.WndProc]
44009F [Controls][Controls.TControl.Perform]
43ECDf [Controls][Controls.TControl.SetVisible]
45F770
76743833 [BaseThreadInitThunk]
```

The current stack trace leading to this error (return addresses):

```
40307A [System][System.@FreeMem]
44091A [Controls][Controls.TControl.Click]
44431B [Controls][Controls.TwinControl.WndProc]
42D959 [StdCtrls][StdCtrls.TButtonControl.WndProc]
44446C [Controls][Controls.DoControlMsg]
44431B [Controls][Controls.TwinControl.WndProc]
45498A [Forms][Forms.TCustomForm.WndProc]
443A43 [Controls][Controls.TwinControl.MainWndProc]
41F31A [Classes][Classes.StdWndProc]
76281A10 [GetMessageW]
```

Current memory dump of 256 bytes starting at pointer address 133DEF8:
46 61 73 74 4D 4D 43 6F 64 [... omitted...]

Not that this is extremely obvious, but it should provide enough information to get you started on chasing down the bug. Notice that without these settings in the memory manager, you'll basically won't see any error and the program keeps running... although you might experience random bugs in case the buffer overrun affects an area of memory in which something else is stored. At that point you can get some weird and very hard to track errors.

As an example, I once saw the partial overwriting of the initial portion of the data of an object, where the class reference is stored. Through this memory corruption the class became undefined and each and every call to one of its virtual functions would badly crash... something very hard to relate to a memory writing operation in a totally different area of the program.

Memory Management on Platforms Other than Windows

Given how memory management works in Object Pascal compilers, it is worth considering some of the options you have in making sure everything is under control.

Before we proceed, it is important to notice that on non-Windows platforms Delphi doesn't use the FastMM4 memory manager, so setting the `ReportMemoryLeaksOnShutdown` global flag to check for memory leaks when the program closes is useless. There is also another reason, which is that there isn't generally a way to close an application on mobile, given apps stay in memory until forcibly removed by the user or the operating system.

On the macOS, iOS and Android platforms, the Object Pascal RTL directly calls the `malloc` and `free` functions of the native `libc` library. One way to monitor memory usage on this platform is to rely on external platform tools. For example on iOS (and macOS) you can use Apple's Instruments tool, which is a complete tracking system monitoring all aspects of your applications running on a physical device.

Tracking Per-Class Allocations

Finally, there is an Object Pascal specific approach for tracking for a specific class rather than memory management at large. The memory allocation for an object, in fact, takes place by calling the `NewInstance` virtual class method, while the cleanup is done by the `FreeInstance` virtual method. These are virtual methods you can override in a specific class to customize the specific memory allocation strategy.

The advantage is you can do this regardless of the constructors (as you can have more than one) and the destructor, clearly separating the memory tracking code from the standard object initialization and finalization code.

While that is a rather extreme corner case (probably only worth doing for some large memory structures) you can override these methods to count the number of objects of a given class that are created and destroyed, calculate the number of active instances, and at the end check that the count goes to zero as expected.

Writing Robust Applications

In this chapter and in many previous chapters of this section, I have covered quite a few techniques focused on writing robust applications and properly managing memory allocation and deallocation.

In this final section of a chapter focused on memory management, I decided to list a few slightly more advanced topics, that augment the earlier coverage. Even if using try-finally blocks and calling destructors have already been covered, the scenarios

388 - 13: Objects and Memory

highlighted here are slightly more complex and involve using multiple language features together.

This is not really an advanced section, but something all Object Pascal developers should really master to be able to write robust applications. Only the last sub-section on pointers and object references is definitely more advanced in scope as it delves into the internal in-memory structure of an object and a class reference.

Constructors, Destructors, and Exceptions

Constructors and destructors can often be a source of problems within applications. Virtual constructors must invariably call their base class constructor *first*. Destructors should generally call their inherited *last*, instead.

note To follow what is good coding practice, you should generally add a base class constructor call in every constructor of your Object Pascal code, even if this is not compulsory and the extra call might be useless (like when calling `TObject.Create`).

In this section I want to specifically focus on what happens when a constructor fails in a classic scenario like:

```
MyObj := TMyClass.Create;
try
  MyObj.DoSomething;
finally
  MyObj.Free;
end;
```

If the object is created and assigned to the `MyObj` variable, the `finally` block takes care of destroying it. But if the `Create` call raises an exception, the `try-finally` block is not entered (*and this is correct!*). When a constructor raises an exception, the corresponding destructor code is *automatically executed* on what might be a partially initialized object. If the constructor creates two sub-objects, for example, those need to be cleared invoking the matching destructor. However this can lead to potential trouble if in the destructor you presume the object was fully initialized..

This is not simple to understand in theory, so let's look to a practical demo in code. The `SafeCode` application project contains a class with a constructor and a destructor that will be generally correct... unless the constructor itself fails:

```
type
  TUnsafeDestructor = class
  private
    FList: TList;
  public
    constructor Create (PositiveNumber: Integer);
```

```

    destructor Destroy; override;
end;

constructor TUnsafeDescructor.Create(PositiveNumber: Integer);
begin
    inherited Create;

    if PositiveNumber <= 0 then
        raise Exception.Create ('Not a positive number');
    FList := TList.Create;
end;

destructor TUnsafeDescructor.Destroy;
begin
    FList.Clear;
    FList.Free;
    inherited;
end;

```

The problem is that in cases where the object has been fully created, the destructor works correctly, but if it is executed when the `FList` field is still set to `nil`, the `Clear` call will raise an “Access violation” exception.

The safe way to write the same code is the following:

```

destructor TUnsafeDescructor.Destroy;
begin
    if assigned (FList) then
        FList.Clear;
        FList.Free;
    inherited;
end;

```

And the moral of the story, again, is never to take for granted in a destructor that the corresponding constructor has fully initialized the object. You can make this assumption for any other method but not for the destructor.

Nested Finally blocks

Finally blocks are probably the most important and common technique to make your programs safe. I don't think this is an advanced topic, but do you use finally all over the place or not? And do you use it properly in border cases, such as nested operations, or do you combine multiple finalization statements in a single finally block? This is a far from perfect code example:

```

procedure TForm1.BtnTryFClick(Sender: TObject);
var
    A1, A2: TAClass;
begin
    A1 := TAClass.Create;

```

390 - 13: Objects and Memory

```
A2 := TAClass.Create;
try
  A1.Whatever := 'one';
  A2.Whatever := 'two';
finally
  A2.Free;
  A1.Free;
end;
end;
```

This is a safer version of the same code (extracted again from the SafeCode application project):

```
procedure TForm1.BtnTryFClick(Sender: TObject);
var
  A1, A2: TAClass;
begin
  A1 := TAClass.Create;
  try
    A2 := TAClass.Create;
    try
      A1.Whatever := 'one';
      A2.Whatever := 'two';
    finally
      A2.Free;
    end;
  finally
    A1.Free;
  end;
end;
```

Dynamic Type Checking

Dynamic casting operations between types in general, and class types in particular is another possible source of pitfalls. Particularly if you don't use the `is` and `as` operators and simply do *hard* casts. Every direct typecast is, in fact, a potential source of error (unless it follows an `is` check).

Typecasting from object to pointers, to and from class references, from objects to interfaces, to and from strings is potentially very dangerous, but in some special circumstances hard to avoid. For example, you might want to save the object reference in the `Tag` property of a component. Another case is, when you save objects in a list of pointers, an old-fashioned `TList` (rather than a type-safe generic list, covered in the next chapter). This is a quite stupid example:

```
procedure TForm1.BtnCastClick(Sender: TObject);
var
  List: TList;
begin
```

```

List := TList.Create;
try
  List.Add(Pointer(Sender));
  List.Add(Pointer(23422));
  // direct cast
  TButton(List[0]).Caption := 'ouch';
  TButton(List[1]).Caption := 'ouch';
finally
  List.Free;
end;
end;

```

Running this code will generally cause an access violation.

note I wrote *generally* because when you access memory at random you never know the actual effect. At times programs simply overwrite memory without causing an immediate error... but you'll have a hard time later on figuring out why some other data got corrupted.

You should avoid similar situations whenever possible, but if you happen to have no alternative what can you do to fix this code? The natural approach would be to use either an `as` safe cast or an `is` type check, like in the following snippets:

```

// as cast
(TObject(List[0]) as TButton).Caption := 'ouch';
(TObject(List[1]) as TButton).Caption := 'ouch';

// is cast
if TObject(List[0]) is TButton then
  TButton(List[0]).Caption := 'ouch';
if TObject(List[1]) is TButton then
  TButton(List[1]).Caption := 'ouch';

```

However, this is *not* the solution, you'll continue to get *Access violations*. The problem is that both `is` and `as` end up calling `TObject.InheritsFrom`, a difficult operation to perform on a number!

The solution? The real solution is to avoid similar situations in the first place (that type of code honestly makes little sense), using a `TObjectList` or some other safe technique (again, see the next chapter for generic container classes). If you are really in for low-level hacks and like *playing* with pointers, you can try to figure out whether a given “*numeric value*” is really a reference to an object or not. This is not a trivial operation, though. There is an interesting side to it, which I'm taking as the excuse for this demo to explain you the internal structure of an object... and of a class reference.

Is this Pointer an Object Reference?

This section explains the internal structure of objects and class references, and goes way beyond the level of the discussion in most of this book. Still, it can provide the more experts readers with some interesting insights, so I decided to keep this material that I wrote in the past for an advanced paper on memory management. Notice also that the specific implementation below is really Windows specific, in terms of memory checks.

There are times when you have pointers around (a pointer is just a numeric value referring to the physical memory location of some data). These pointers might actually be references to objects, and you generally know when they are and use them as such. But every time you do a low-level cast you are really on the verge of screwing up an entire program. There are techniques, to make this type of pointer management a little safer, even if not 100 percent guarantee them.

The starting point you might want to consider before working with a pointer is whether it is actually a legal pointer or not. The `Assigned` function only checks whether a pointer is not `nil`, which doesn't help in this case. However, the little-known `FindInstance` function of the Object Pascal RTL (in the `System` unit, available on the Windows platform) returns the base address of the heap block including the object passed as parameter, or zero if the pointer refers to an invalid page (preventing rather infrequent but extremely hard to track memory page errors). If you take a number almost at random, it is likely it won't refer to a valid memory page.

This is a good starting point, but we can do better, as this won't help if the value is a string reference or any other valid pointer and not an object reference. Now how do you know if a pointer is actually a reference to an object? I've come up with the following empirical test. The first 4 bytes of an object are the pointer to its class. If you consider the internal data structure of a class reference, it has in its `vmtSelfPtr` position a pointer to itself. This is roughly depicted in the image in Figure 13.7.

Figure 13.7: An approximate representation of the internal structure of objects and class references.



In other words, by dereferencing the value at a memory location `vmtSelfPtr` bytes from the class reference pointer (this is a negative offset, lower in memory) you should get the same class reference pointer again. Moreover, in the internal data

structure of a class reference, you can read the instance size information (at the `vmtInstancesSize` position) and see if there is a *reasonable* number in there. Here is the actual code:

```
function IsPointerToObject (Address: Pointer): Boolean;
var
  ClassPointer, VmtPointer: PChar;
  Instsize: Integer;
begin
  Result := False;
  if (FindHInstance (Address) > 0) then
    begin
      VmtPointer := PChar(Address^);
      ClassPointer := VmtPointer + vmtSelfPtr;
      if Assigned (VmtPointer) and
        (FindHInstance (VmtPointer) > 0) then
        begin
          Instsize := (PInteger(
            VmtPointer + vmtInstancesSize))^;
          // check self pointer and "reasonable" instance size
          if (Pointer(Pointer(ClassPointer)^) =
            Pointer(VmtPointer)) and
            (Instsize > 0) and (Instsize < 10000) then
            Result := True;
        end;
    end;
  end;
end;
```

Having this function at hand, in the previous `SafeCode` application project we can add a pointer-to-object check before making a safe cast:

```
if IsPointerToObject (List[0]) then
  (TObject(list[0]) as TButton).Caption := 'ouch';
if IsPointerToObject (List[1]) then
  (TObject(list[1]) as TButton).Caption := 'ouch';
```

The same idea can also be applied directly to class references, also for implementing safe-casts among them. Again, it is best to try to avoid similar problems in the first place by writing safer and cleaner code, but in case you can't avoid it this function might come in handy. In any case, this section should have explained a little of the internals of these system data structures.

394 - 13: Objects and Memory

part iii: advanced features

Now that we've delved into the language foundations and into the object-oriented programming paradigm, it is time to discover some of the latest and more advanced features of the Object Pascal language. Generics, anonymous methods, and reflection open up to developing code using new paradigms that extend object-oriented programming in significant ways.

Some of these more advanced language features, in fact, let developers embrace new ways of writing code, offering even more type and code abstractions, and allowing for a more dynamic approach to coding using reflection to its fullest potential.

The last part of the section will expand on these language features by offering an overview of core run-time library elements, which are so core to the Object Pascal development model to make the distinction between language and library quite blurred. We'll inspect, for example, the `TObject` class that, as we saw earlier, is the base class of all classes you write: far too prominent a role to be confined to a library implementation detail.

Chapters of Part III

Chapter 14: Generics

Chapter 15: Anonymous Methods

Chapter 16: Reflection and Attributes

Chapter 17: The TObject Class

Chapter 18: The Run Time Library

14: generics

The strong type checking provided by Object Pascal is useful for improving the correctness of the code, a topic I've stressed a lot in this book. Strong type checking, though, can also be a nuisance, as you might want to write a procedure or a class that can act similarly on different data types. This issue is addressed by a feature of the Object Pascal language, also available in similar languages like C# and Java, called *generics*.

The concept of generic or template classes actually comes from the C++ language. This is what I wrote in 1994 in a book about C++:

You can declare a class without specifying the type of one or more data members: this operation can be delayed until an object of that class is actually declared. Similarly, you can define a function without specifying the type of one or more of its parameters until the function is called.

note The text is extracted from the book “Borland C++ 4.0 Object-Oriented Programming” I wrote with Steve Tendon in the early 90ies.

This chapter delves into the topic, starting with the foundations but also covering some advanced usage scenarios, and even indicating how generics can even be applied to standard visual programming.

Generic Key-Value Pairs

As a first example of a generic class, I've implemented a key-value pair data structure. The first code snippet below shows the data structure written in a traditional fashion, with an object used to hold the value:

```
type
  TKeyValue = class
    private
      FKey: string;
      FValue: TObject;
      procedure SetKey(const value: string);
      procedure SetValue(const value: TObject);
    public
      property Key: string read FKey write SetKey;
      property Value: TObject read FValue write SetValue;
  end;
```

To use this class you can create an object, set its key and value, and use it, as in the following snippets of various methods of the main form of the `keyvalueClassic` application project:

```
// FormCreate
Kv := TKeyValue.Create;

// Button1Click
Kv.Key := 'mykey';
Kv.Value := Sender;

// Button2Click
Kv.Value := self; // the form

// Button3Click
ShowMessage('[' + Kv.Key + ', ' + Kv.Value.ClassName + ']');
```

What if you need a similar class, holding an Integer rather than an object? Well, either you make a very unnatural (and dangerous) type cast, or you create a new and separate class to hold a string key with a numeric value. Although copy and paste of the original class might sound a solution, you end up with two copies for a very similar piece of code, you are going against good programming principles... and you'll have to update with new features or correct the same bugs two, or three or twenty times.

Generics make it possible to use a much broader definition for the value, writing a single generic class. Once you instantiate the key-value generic class, it becomes a specific class, tied to a given data type. So you still end up with two, or three, or twenty classes compiled into your application, but you have a single source code definition for all of them, still replying on proper string type checking and without a

runtime overhead. But I'm getting ahead of myself: let's start with the syntax used to define the generic class:

```
type
  TKeyValue<T> = class
  private
    FKey: string;
    FValue: T;
    procedure SetKey(const Value: string);
    procedure SetValue(const value: T);
  public
    property Key: string read FKey write SetKey;
    property Value: T read FValue write SetValue;
end;
```

In this class definition, there is one unspecified type, indicated by the placeholder τ , placed within angle brackets. The symbol τ is frequently used by convention, but as far as the compiler is concerned you can use just any symbol you like. Using τ generally makes the code more readable when the generic class uses only one parametric type; in case the class needs multiple parametric types it is common to name them according to their actual role, rather than using a sequence of letters (τ , u , v) as it happened in C++ during the early days.

note “T” has been the standard name, or placeholder, for a generic type since the days the C++ language introduced *templates* in the early 1990s. Depending on the authors, the “T” stands for either “Type” or “Template type”.

The generic `TKeyValue<T>` class uses the unspecified type as the type of one of its two fields, the property value, and the setter method parameter. The methods are defined as usual, but notice that regardless of the fact they have to do with the generic type, their definition contains the complete name of the class, including the generic type:

```
procedure TKeyValue<T>.SetKey(const Value: string);
begin
  FKey := Value;
end;

procedure TKeyValue<T>.SetValue(const value: T);
begin
  FValue := value;
end;
```

To use the class, instead, you have to fully qualify it, providing the actual type of the generic type. For example, you can now declare a key-value object hosting buttons as values by writing:

```
var
  Kv: TKeyValue<TButton>;
```


400 - 14: Generics

The full name is required also when creating an instance, because this is the actual type name (while the generic, uninstantiated type name is like a type construction mechanism).

Using a specific type of the value of the key-value pair makes the code much more robust, as you can now only add `TButton` (or derived) objects to the key-value pair and can use the various methods of the extracted object. These are some snippets from the main form of the `KeyValueGeneric` application project:

```
// FormCreate
Kv := TKeyValue<TButton>.Create;

// Button1Click
Kv.Key := 'mykey';
Kv.Value := Sender as TButton;

// Button2Click
Kv.Value := Sender as TButton; // was "self"

// Button3Click
ShowMessage ('[' + Kv.Key + ', ' + Kv.Value.Name + ']');
```

When assigning a generic object in the previous version of the code we could add either a button or a form, now we can use only button, a rule enforced by the compiler. Likewise, rather than a generic `kv.Value.ClassName` in the output we can use the component `Name`, or any other property of the `TButton` class.

Of course, we can also mimic the original program by declaring the key-value pair with an object type, like:

```
var
  Kvo: TKeyValue<TObject>;
```

In this version of the generic key-value pair class, we can add any object as value. However, we won't be able to do much on the extracted objects, unless we cast them to a more specific type. To find a good balance, you might want to go for something in between specific buttons and any object, requesting the value to be a component:

```
var
  Kvc: TKeyValue<TComponent>;
```

You can see corresponding code snippets in the same `KeyValueGeneric` application project. Finally, we can also create an instance of the generic key-value pair class that doesn't store object values, but rather plain integers:

```
var
  Kvi: TKeyValue<Integer>;
begin
  Kvi := TKeyValue<Integer>.Create;
  try
    Kvi.Key := 'object';
    Kvi.Value := 100;
```

```

    Kvi.Value := Left;
    ShowMessage ('[' + Kvi.Key + ', ' +
        IntToStr (Kvi.Value) + ']');
finally
    Kvi.Free;
end;

```

Inline Variables and Generics Type Inference

When you are declaring a variable of a generic type, the declaration can be fairly long. As you create an object of that type, you have to repeat the same declaration. That is, unless you take advantage of inline variable declarations and their ability to infer the variable type. The last code fragment above can be written as:

```

begin
    var Kvi := TKeyValue<Integer>.Create;
    try
        ...

```

In this code you don't have to repeat the full generic type declaration twice. This is particularly handy when using containers, as we'll see later.

Type Rules on Generics

When you declare an instance of a generic type, this type gets a specific version, which is enforced by the compiler in all subsequent operations. So if you have a generic class like:

```

type
    TSimpleGeneric<T> = class
        value: T;
    end;

```

as you declare a specific object with a given type, you cannot assign a different type to the `value` field. Given the following two objects, some of the assignments below (part of the `TypeCompRules` application project) are incorrect:

```

var
    Sg1: TSimpleGeneric<string>;
    Sg2: TSimpleGeneric<Integer>;
begin
    Sg1 := TSimpleGeneric<string>.Create;
    Sg2 := TSimpleGeneric<Integer>.Create;

    Sg1.value := 'foo';
    Sg1.value := 10; // Error
    // E2010 Incompatible types: 'string' and 'Integer'

```

```
Sg2.value := 'foo'; // Error
// E2010 Incompatible types: 'Integer' and 'string'
Sg2.value := 10;
```

Once you define a specific type in the generic declaration, this is enforced by the compiler, as you should expect from a strongly-typed language like Object Pascal. Type checking is also in place for generic objects as a whole. As you specify the generic parameter for an object, you cannot assign to it a similar generic type based on a different and incompatible type instance. If this seems confusing, an example should help clarifying:

```
Sg1 := TSimpleGeneric<Integer>.Create; // Error
// E2010 Incompatible types:
// 'TSimpleGeneric<System.string>'
// and 'TSimpleGeneric<System.Integer>'
```

As we'll see in the section “Generic Types Compatibility Rules” in this peculiar case the type compatibility rule is by structure and not by type name. You cannot assign a different and incompatible type to a generic type once it has been declared.

Generics in Object Pascal

In the previous example we have seen how you can define and use a generic class in Object Pascal. I decided to introduce this feature with an example before delving into the technicalities, which are quite complex and very important at the same time. After covering generics from a language perspective we'll get back to more examples, including the use and definition of generic container classes, one of the main uses of this technique in the language.

We have seen that when you define a class you can add in an extra “parameter” within angle brackets to hold the place of a type to be provided later:

```
type
  TMyClass<T> = class
  ...
end;
```

The generic type can be used as the type of a field (as I did in the previous example), as the type of a property, as the type of a parameter or return value of a function, and more. Notice that it is not compulsory to use the type for a local field (or array), as there are cases in which the generic type is used only as a result, a parameter, or is not used in the declaration of the class, but only in the definition of some of its methods.

This form of extended or *generic* type declaration is not only available for classes but also for records (that as I covered in Chapter 5, can also have methods, properties, and overloaded operators). A generic class can also have multiple parameterized types, as in following case in which you can specify an input parameter and a return value of a different type for a method:

```
type
  TPWGeneric<TInput,TReturn> = class
    public
      function AnyFunction (Value: TInput): TReturn;
    end;
```

The implementation of generics in Object Pascal, like in other static languages is not based on runtime support. It is handled by the compiler and the linker, leaving almost nothing to the runtime mechanism. Unlike virtual function calls, which are bound at runtime, generic class methods are generated once for each generic type you instantiate, and are generated at compile time! We'll see the possible drawbacks of this approach, but on the positive side it implies that generic classes are as efficient as plain classes, or even more efficient as the need for runtime checks is reduced. Before we look at some of the internals, though, let me focus on some very significant rules which break the traditional Pascal language type compatibility rules.

Generic Types Compatibility Rules

In traditional Pascal and in Object Pascal the core type compatibility rules are based on type name equivalence. In other words, two variables are type compatible only if their type name is the same, regardless of the actual data structure to which they refer.

This is a classic example of type incompatibility with static arrays (part of the `Type-CompRules` application project):

```
type
  TArrayOf10 = array [1..10] of Integer;

procedure TForm30.Button1Click(Sender: TObject);
var
  Array1: TArrayOf10;
  Array2: TArrayOf10
  Array3, Array4: array [1..10] of Integer;
begin
  Array1 := Array2;
  Array2 := Array3; // Error
  // E2010 Incompatible types: 'TArrayOf10' and 'Array'
```

404 - 14: Generics

```
Array3 := Array4;  
Array4 := Array1; // Error  
// E2010 Incompatible types: 'Array' and 'TArrayOf10'  
end;
```

As you can see in the code above, all four arrays are structurally identical. However, the compiler will let you assign only those that are type compatible, either because their type has the same explicit name (like `TArrayOf10`) or because they have the same implicit (or compiler generated, type name, as the two arrays declared in a single statement.

This type compatibility rule has very limited exceptions, like those related to derived classes. Another exception to the rule, and a significant one, is type compatibility for generic types, which is probably also used internally by the compiler to determine when to *generate* a new type from the generic one, with all of its methods.

The new rule states that generic types are compatible when they share the same generic class definition and instance type, regardless of the type name associated with this definition. In other words, the full name of the generic type instance is a combination of the generic type and the instance type.

In the following example the four variables are all type compatible:

```
type  
  TGenericArray<T> = class  
    AnArray: array [1..10] of T;  
  end;  
  
  TIntGenericArray = TGenericArray<Integer>;  
  
procedure TForm30.Button2Click(Sender: TObject);  
var  
  Array1: TIntGenericArray;  
  Array2: TIntGenericArray;  
  Array3, Array4: TGenericArray<Integer>;  
begin  
  Array1 := TIntGenericArray.Create;  
  Array2 := Array1;  
  Array3 := Array2;  
  Array4 := Array3;  
  Array1 := Array4;  
end;
```

Generic Methods for Standard Classes

While the use of generics types to define classes is likely the most common scenario, generic types can also be used in non-generic classes. In other words, a regular class can have a generic method. In this case, you don't specific a specific type for the

generic placeholder when you create an instance of the class, but also when you invoke the method. Here is an example class with a generic method from the `GenericMethod` application project:

```
type
  TGenericFunction = class
    public
      function WithParam <T> (T1: T): string;
    end;
```

note When I first wrote this code, probably with a reminiscence of my C++ days, I wrote the parameter as `(t: T)`. Needless to say in a case insensitive language like Object Pascal, this is not a great idea. The compiler will actually let it go but issue errors every time you refer to the generic type `T`.

There isn't much you can do inside a similar class method (at least unless you use constraints, covered later in this chapter), so I wrote some code using special generic type functions (again covered later) and a special function to convert the type to a string, which it is not relevant to discuss here:

```
function TGenericFunction.WithParam<T>(T1: T): string;
begin
  Result := GetTypeName (TypeInfo (T));
end;
```

As you can see this method doesn't even use the actual value passed as the parameter, but only grabs some type information. Again, not knowing the type of `t1` at all makes it fairly complex to use it in code.

You can call various versions of this “global generic function” as follows:

```
var
  Gf: TGenericFunction;
begin
  Gf := TGenericFunction.Create;
  try
    Show (Gf.WithParam<string>('foo'));
    Show (Gf.WithParam<Integer>(122));
    Show (Gf.WithParam('hello'));
    Show (Gf.WithParam(122));
    Show (Gf.WithParam(Button1));
    Show (Gf.WithParam<TObject>(Button1));
  finally
    Gf.Free;
  end;
```

All of the calls above are correct, as the parametric type can be implicit in these calls. Notice the generic type is displayed (as specified or inferred) and not the actual type of the parameter, which explains this output:

```
string
Integer
string
```

406 - 14: Generics

```
ShortInt  
TButton  
TObject
```

If you call the method without indicating the type between angle brackets, the actual type is inferred from the parameter's type. If you call the method with a type and a parameter, the parameter's type must match the generic type declaration. So the three lines below won't compile:

```
Show (Gf.WithParam<Integer>('foo'));  
Show (Gf.WithParam<String> (122));  
Show (Gf.WithParam<TButton>(self));
```

Generic Type Instantiation

Notice this is a rather advanced section focusing on some of the internals of generics and their potential optimization. Good for a second read, not if this is the first time you are looking into generics.

With the exception of some optimizations, every time you instantiate a generic type, whether in a method or in a class, a new type is generated by the compiler. This new type shares no code with different instances of the same generic type (or different versions of the same method).

Let's look at an example (which is part of the `GenericCodeGen` application project). The program has a generic class defined as:

```
type  
  TSampleClass <T> = class  
    private  
      data: T;  
    public  
      procedure One;  
      function ReadT: T;  
      procedure SetT (value: T);  
    end;
```

The three methods are implemented as follows (notice that the `One` method is absolutely independent from the generic type):

```
procedure TSampleClass<T>.One;  
begin  
  Form30.Show ('OneT');  
end;  
  
function TSampleClass<T>.ReadT: T;  
begin  
  Result := data;  
end;
```

```

procedure TSampleClass<T>.SetT(value: T);
begin
    data := value;
end;

```

Now the main program uses the generic type mostly to figure out the in-memory address of its methods once an instance is generated (by the compiler). This is the code

```

procedure TForm30.Button1Click(Sender: TObject);
var
    T1: TSampleClass<Integer>;
    T2: TSampleClass<string>;
begin
    T1 := TSampleClass<Integer>.Create;
    T1.SetT (10);
    T1.One;

    T2 := TSampleClass<string>.Create;
    T2.SetT ('hello');
    T2.One;

    Show ('T1.SetT: ' +
        IntToHex (PInteger(@TSampleClass<Integer>.SetT)^, 8));
    Show ('T2.SetT: ' +
        IntToHex (PInteger(@TSampleClass<string>.SetT)^, 8));

    Show ('T1.One: ' +
        IntToHex (PInteger(@TSampleClass<Integer>.One)^, 8));
    Show ('T2.One: ' +
        IntToHex (PInteger(@TSampleClass<string>.One)^, 8));
end;

```

The result is something like this (the actual values will vary):

```

T1.SetT: C3045089
T2.SetT: 51EC8B55
T1.One: 4657F0BA
T2.One: 46581CBA

```

As I anticipated, not only does the `SetT` method get a different version in memory generated by the compiler for each data type used, but even the `One` method does, despite the fact they are all identical.

Moreover, if you redeclare an identical generic type, you'll get a new set of implementation functions. Similarly, the same instance of a generic type used in different units forces the compiler to generate the same code over and over, possibly causing significant code bloat. For this reason if you have a generic class with many methods that don't depend on the generic type, it is recommended to define a base non-generic class with those common methods and an inherited generic class with the generic methods: this way the base class methods are only compiled and included in the executable once.

note There is currently compiler, linker, and low-level RTL work being done to reduce the size increase caused by generics in scenarios like those outlined in this section. See for example the considerations in <http://delphisorcery.blogspot.it/2014/10/new-language-feature-in-xe7.html>.

Generic Type Functions

The biggest problem with the generic type definitions we have seen so far is that there is very little you can do with elements of the generic class type. There are two techniques you can use to overcome this limitation. The first is to make use of the few special functions of the run-time library that specifically support generic types; the second (and much more powerful) is to define generic classes with constraints on the types you can use.

I'll focus on the first technique in this section and on constraints in the next section. As I mentioned, there are some RTL functions that work on the parametric type (τ) of generic type definition:

- `Default(τ)` is actually a new function introduced along with generics that returns the empty or “zero value” or null value for the current type; this can be zero, an empty string, `nil`, and so on; the zero-initialized memory has the same value of a global variable of the same type (differently from local variables, in fact, global ones are initialized to “zero” by the compiler);
- `TypeInfo(τ)` returns the pointer to the run-time information for the current version of the generic type; you'll find a lot more information about type information in Chapter 16;
- `SizeOf(τ)` returns memory size of the type in bytes (which in case of a reference type like a string or an object would be the size of the reference, that is 4 bytes for a 32-bit compiler and 8 bytes for a 64-bit compiler).
- `IsManagedType(τ)` indicates if the type is managed in memory, as happens for strings and dynamic arrays
- `HasWeakRef(τ)` is tied to ARC-enabled compilers, and indicates whether the target type has weak references, requiring specific memory management support
- `GetTypeKind(τ)` is a shortcut for accessing the type kind from the type information; which is a slightly higher level type definition than the one returned by `TypeInfo`.

note All of these methods return compiler evaluated constants rather than calling actual functions at run-time. The importance of this is not in the fact these operations are very fast, but that this makes it possible for the compiler and the linker to optimize the generated code, removing unused branches. If you have a case or an if statement based on the return value of one of these functions, the compiler can figure out that for a given type only one of the branches is going to be executed, removing the useless code. When the same generic method is compiled for a different type, it might end up using a different branch, but again the compiler can figure out up front and optimize the size of the method.

The `GenericTypeFunc` application project has a generic class showing the three generic type functions in action:

```

type
  TSampleClass <T> = class
    private
      FData: T;
    public
      procedure Zero;
      function GetDataSize: Integer;
      function GetDataName: string;
    end;

function TSampleClass<T>.GetDataSize: Integer;
begin
  Result := SizeOf (T);
end;

function TSampleClass<T>.GetDataName: string;
begin
  Result := GetTypeName (TypeInfo (T));
end;

procedure TSampleClass<T>.Zero;
begin
  FData := Default (T);
end;

```

In the `GetDataName` method I used the `GetTypeName` function (of the `System.TypeInfo` unit) rather than directly accessing the data structure because it performs the proper conversion from the encoded string value holding the type name.

Given the declaration above, you can compile the following test code, that repeats itself three times on three different generic type instances. I've omitted the repeated code, but show the statements used to access the data field, as they change depending on the actual type:

```

var
  T1: TSampleClass<Integer>;
  T2: TSampleClass<string>;
  T3: TSampleClass<double>;
begin

```

410 - 14: Generics

```
T1 := TSampleClass<Integer>.Create;  
T1.Zero;  
Show ('TSampleClass<Integer>');  
Show ('data: ' + IntToStr (T1.FData));  
Show ('type: ' + T1.GetDataName);  
Show ('size: ' + IntToStr (T1.GetDataSize));  
  
T2 := TSampleClass<string>.Create;  
...  
Show ('data: ' + T2.FData);  
  
T3 := TSampleClass<double>.Create;  
...  
Show ('data: ' + FloatToStr (T3.FData));
```

Running this code (from the GenericTypeFunc application project) produces the following output:

```
TSampleClass<Integer>  
data: 0  
type: Integer  
size: 4  
TSampleClass<string>  
data:  
type: string  
size: 4  
TSampleClass<double>  
data: 0  
type: Double  
size: 8
```

Notice that you can use the generic type functions also on specific types, outside of the context of generic classes. For example, you can write:

```
var  
  I: Integer;  
  s: string;  
begin  
  I := Default (Integer);  
  Show ('Default Integer': + IntToStr (I));  
  
  s := Default (string);  
  Show ('Default String': + s);  
  
  Show ('TypeInfo String': +  
    GetTypeNames (TypeInfo (string)));
```

This is the trivial output:

```
Default Integer: 0  
Default String:  
TypeInfo String: string
```

note You cannot apply the `TypeInfo` call to a variable, like `TypeInfo(s)` in the code above, but only to a data type.

Class Constructors for Generic Classes

A very interesting case arises when you define a class constructor for a generic class. In fact, one such constructor is generated by the compiler and called for each generic class instance, that is, for each actual type defined using the generic template. This is quite interesting, because it would be very complex to execute initialization code for each actual instance of the generic class you are going to create in your program without a class constructor.

As an example, consider a generic class with some class data. You'll get an instance of this class data for each generic class instance. If you need to assign an initial value to this class data, you cannot use the unit initialization code, as in the unit defining the generic class you don't know which actual classes you are going to need.

The following is a bare bones example of a generic class with a class constructor used to initialize the `DataSet` class field, taken from the `GenericClassCtor` application project:

```
type
  TGenericwithClassCtor <T> = class
    private
      FData: T;
    procedure SetData(const Value: T);
    public
      class constructor Create;
      property Data: T read FData write SetData;
      class var
        DataSet: Integer;
      end;
end;
```

This is the code of the generic class constructor, which uses an internal string list (see the full source code for implementation details) for keeping track of which class constructors are actually called:

```
class constructor TGenericwithClassCtor<T>.Create;
begin
  DataSet := SizeOf (T);
  ListSequence.Add(ClassName);
end;
```

The demo program creates and uses a couple of instances of the generic class, and also declares the data type for a third, which is removed by the linker:

```
var
```

412 - 14: Generics

```
GenInt: TGenericWithClassCtor <SmallInt>;  
GenStr: TGenericWithClassCtor <string>;  
type  
  TGenDouble = TGenericWithClassCtor <Double>;
```

If you ask the program to show the contents of the `ListSequence` string list, you'll see only the types that have actually been initialized:

```
TGenericWithClassCtor<System.SmallInt>  
TGenericWithClassCtor<System.string>
```

However, if you create generic instances based on the same data type in different units, the linker might not work as expected and you'll have multiple calls to the same generic class constructor (or, to be more precise, two generic class constructors for the same type).

note It is not easy to address a similar problem. To avoid a repeated initialization, you might want to check if the class constructor has already been executed. In general, though, this problem is part of a more comprehensive limitation of generic classes and the linker's inability to optimize them.

I've added a procedure called `Useless` in the secondary unit of this example that, when un-commented, will highlight the problem, with an initialization sequence like:

```
TGenericWithClassCtor<System.string>  
TGenericWithClassCtor<System.SmallInt>  
TGenericWithClassCtor<System.string>
```

Generic Constraints

As we have seen, there is very little you can do in the methods of your generic class over the generic type value. You can pass it around (that is, assign it) and perform the limited operations allowed by the generic type functions I've just covered.

To be able to perform some actual operations of the generic type of class, you generally have to place a constraint on it. For example, if you limit the generic type to be a class, the compiler will let you call all of the `TObject` methods on it. You can also further constrain the class to be part of a given hierarchy or to implement a specific interface, making it possible to call the class or interface method on an instance of the generic type.

Class Constraints

The simplest constraint you can adopt is a class constraint. To use it, you can declare generic type as:

```
type
  TSampleClass <T: class> = class
```

By specifying a class constraint you indicate that you can use only object types as generic types. With the following declaration (taken from the `ClassConstraint` application project):

```
type
  TSampleClass <T: class> = class
private
  FData: T;
public
  procedure One;
  function ReadT: T;
  procedure SetT (T1: T);
end;
```

you can create the first two instances but not the third:

```
sample1: TSampleClass<TButton>;
sample2: TSampleClass<TStrings>;
sample3: TSampleClass<Integer>; // Error
```

The compiler error caused by this last declaration would be:

```
E2511 Type parameter 'T' must be a class type
```

What's the advantage of indicating this constraint? In the generic class methods you can now call any `TObject` method, including virtual ones! This is the one method of the `TSampleClass` generic class:

```
procedure TSampleClass<T>.One;
begin
  if Assigned (FData) then
    begin
      Form30.Show ('ClassName: ' + FData.ClassName);
      Form30.Show ('Size: ' + IntToStr (FData.InstanceSize));
      Form30.Show ('ToString: ' + FData.ToString);
    end;
  end;
```

note Two comments here. The first is that `InstanceSize` returns the actual size of the object, unlike the generic `SizeOf` function we used earlier, which returns the size of the reference type. Second, notice the use of the `ToString` method of the `TObject` class.

You can play with the program to see its actual effect, as it defines and uses a few instances of the generic type, as in the following code snippet:

414 - 14: Generics

```
var
  Sample1: TSampleClass<TButton>;
begin
  Sample1 := TSampleClass<TButton>.Create;
  try
    Sample1.SetT (Sender as TButton);
    Sample1.One;
  finally
    Sample1.Free;
  end;
```

Notice that by declaring a class with a customized `ToString` method, this version will get called when the data object is of the specific type, regardless of the actual type provided to the generic type. In other words, if you have a `TButton` descendant like:

```
type
  TMyButton = class (TButton)
  public
    function ToString: string; override;
  end;
```

You can pass this object as value of a `TSampleClass<TButton>` or define a specific instance of the generic type, and in both cases calling one ends up executing the specific version of `ToString`:

```
var
  Sample1: TSampleClass<TButton>;
  Sample2: TSampleClass<TMyButton>;
  Mb: TMyButton;
begin
  ...
  Sample1.SetT (Mb);
  Sample1.One;
  Sample2.SetT (Mb);
  Sample2.One;
```

Similarly to a class constraint, you can have a record constraint, declared as:

```
type
  TSampleRec <T: record> = class
```

However, there is very little that different records have in common (there is no common ancestor), so this declaration is somewhat limited.

Specific Class Constraints

If your generic class needs to work with a specific subset of classes (a specific hierarchy), you might want to resort to specifying a constraint based on a given base class. For example, if you declare:

```
type
  TCompClass <T: TComponent> = class
```

instances of this generic class can be applied only to component classes, that is, any `TComponent` descendant class. This let's you have a very specific generic type (yes, it sounds odd, but that's what it really is) and the compiler will let you use all of the methods of the `TComponent` class while working on the generic type.

If this seems extremely powerful, think twice. If you consider what you can achieve with inheritance and type compatibility rules, you might be able to address the same problem using traditional object-oriented techniques rather than having to use generic classes. I'm not saying that a specific class constraint is never useful, but it is certainly not as powerful as a higher-level class constraint or (something I find very interesting) an interface-based constraint.

Interface Constraints

Rather than constraining a generic class to a given class, it is generally more flexible to accept as type parameter only classes implementing a given interface. This makes it possible to call the interface on instances of the generic type. This use of interface constraints for generics is also very common in the C# language. Let me start by showing you an example (from the `IntfConstraint` application project). First, we need to declare an interface:

```
type
  IGetValue = interface
    ['{60700EC4-2CDA-4CD1-A1A2-07973D9D2444}']
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
    property Value: Integer read GetValue write SetValue;
end;
```

Next, we can define a class implementing it:

```
type
  TGetValue = class (TSingletonImplementation, IGetValue)
  private
    FValue: Integer;
  public
    constructor Create (Value: Integer = 0);
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
end;
```

Things start to get interesting in the definition of a generic class limited to types that implement the given interface:

```
type
  TIntfClass <T: IGetValue> = class
```


416 - 14: Generics

```
private
  FVal1, FVal2: T; // or IGetValue
public
  procedure Set1 (Val: T);
  procedure Set2 (Val: T);
  function GetMin: Integer;
  function GetAverage: Integer;
  procedure IncreaseByTen;
end;
```

Notice that in the code of the generic methods of this class we can write, for example:

```
function TInftClass<T>.GetMin: Integer;
begin
  Result := Min (FVal1.GetValue, FVal2.GetValue);
end;

procedure TInftClass<T>.IncreaseByTen;
begin
  FVal1.SetValue (FVal1.GetValue + 10);
  FVal2.Value := FVal2.Value + 10;
end;
```

With all these definitions, we can now use the generic class as follows:

```
procedure TFormIntfConstraint.BtnValueClick(
  Sender: TObject);
var
  IClass: TInftClass<TGetValue>;
begin
  IClass := TInftClass<TGetValue>.Create;
  try
    IClass.Set1 (TGetValue.Create (5));
    IClass.Set2 (TGetValue.Create (25));
    Show ('Average: ' + IntToStr (IClass.GetAverage));
    IClass.IncreaseByTen;
    Show ('Min: ' + IntToStr (IClass.GetMin));
  finally
    IClass.val1.Free;
    IClass.val2.Free;
    IClass.Free;
  end;
end;
```

To show the flexibility of this generic class, I've created another totally different implementation for the interface:

```
type
  TButtonValue = class (TButton, IGetValue)
  public
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
    class function MakeTButtonValue (Owner: TComponent;
      Parent: TwinControl): TButtonValue;
```

```

    end;

function TButtonValue.GetValue: Integer;
begin
    Result := Left;
end;

procedure TButtonValue.SetValue(value: Integer);
begin
    Left := value;
end;

```

The class function (not listed in the book) creates a button within a Parent control in a random position and is used in the following sample code:

```

procedure TFormIntfConstraint.BtnValueButtonClick(
    Sender: TObject);
var
    IClass: TInftClass<TButtonValue>;
begin
    IClass := TInftClass<TButtonValue>.Create;
    try
        IClass.Set1 (TButtonValue.MakeTButtonValue (
            self, ScrollBox1));
        IClass.Set2 (TButtonValue.MakeTButtonValue (
            self, ScrollBox1));
        Show ('Average: ' + IntToStr (IClass.GetAverage));
        Show ('Min: ' + IntToStr (IClass.GetMin));
        IClass.IncreaseByTen;
        Show ('New Average: ' + IntToStr (IClass.GetAverage));
    finally
        IClass.Free;
    end;
end;

```

Interface References vs. Generic Interface Constraints

In the last example I have defined a generic class that works with any object implementing a given interface. I could have obtained a similar effect by creating a standard (non-generic) class based on interface references. In fact, I could have defined a class like (again part of the `IntfConstraint` application project):

```

type
    TPlainInftClass = class
    private
        FVal1, FVal2: IGetValue;
    public
        procedure Set1 (Val: IGetValue);
        procedure Set2 (Val: IGetValue);

```

418 - 14: Generics

```
function GetMin: Integer;  
function GetAverage: Integer;  
procedure IncreaseByTen;  
end;
```

What is different between these two approaches? A first difference is that in the class above you can pass two objects of different types to the setter methods, provided their classes both implement the given interface, while in the generic version you can pass only objects of the given type (to any given instance of the generic class). So the generic version is more *conservative* and strict in terms of type checking.

In my opinion, the key difference is that using the interface-based version means having Object Pascal's reference counting mechanism in action, while using the generic version the class is dealing with plain objects of a given type and reference counting is not involved.

Moreover, the generic version could have multiple constraints, like a constructor constraint and lets you use the various generic-functions (like asking for the actual type of the generic type), something you cannot do when using an interface. (When you are working with an interface, in fact, you have no way to refer to the base `TObject` methods).

In other words, using a generic class with an interface constraint makes it possible to have the benefits of interfaces without their nuisances. Still, it is worth noticing that in most cases the two approaches would be equivalent, and in others the interface-based solution would be more flexible.

Default Constructor Constraint

There is another possible generic type constraint, called default constructor or parameterless constructor. If you need to invoke the default constructor to create a new object of the generic type (for example for filling a list) you can use this constraint. In theory (and according to the documentation), the compiler should let you use it only for those types with a default constructor. In practice, if a default constructor doesn't exist, the compiler will let it go and call the default constructor of `TObject`.

A generic class with a constructor constraint can be written as follows (this one is extracted by the `IntfConstraint` application project):

```
type  
  TConstrClass <T: class, constructor> = class  
  private  
    FVal: T;
```

```

public
  constructor Create;
  function Get: T;
end;

```

note You can also specify the constructor constraint without the class constraint, as the former probably implies the latter. Listing both of them makes the code more readable.

Given this declaration, you can use the constructor to create a generic internal object, without knowing its actual type up front, and write:

```

constructor TConstrClass<T>.Create;
begin
  FVal := T.Create;
end;

```

How can we use this generic class and what are the actual rules? In the next example I have defined two classes, one with a default (parameterless) constructor, the second with a single constructor having one parameter:

```

type
  TSimpleConst = class
    public
      FValue: Integer;
      constructor Create; // set value to 10
    end;

  TParamConst = class
    public
      FValue: Integer;
      constructor Create (I: Integer); // set value to I
    end;

```

As I mentioned earlier, in theory you should only be able to use the first class, while in practice you can use both:

```

var
  ConstructObj: TConstrClass<TSimpleCost>;
  ParamCostObj: TConstrClass<TParamCost>;
begin
  ConstructObj := TConstrClass<TSimpleCost>.Create;
  Show ('value 1: ' + IntToStr (ConstructObj.Get.FValue));

  ParamCostObj := TConstrClass<TParamCost>.Create;
  Show ('value 2: ' + IntToStr (ParamCostObj.Get.FValue));

```

The output of this code is:

```

value 1: 10
value 2: 0

```

420 - 14: Generics

In fact, the second object is never initialized. If you debug the application trace into the code you'll see a call to `TObject.Create` (which I consider wrong). Notice that if you try calling directly:

```
| with TParamConst.Create do
```

the compiler will (correctly) raise the error:

```
| [DCC Error] E2035 Not enough actual parameters
```

note Even if a direct call to `TParamConst.Create` will fail at compile time (as explained here), a similar call using a class reference or any other form of indirection will succeed, which probably explains the behavior of the effect of the constructor constraint.

Constraints Summary and Combining Them

As there are so many different constraints you can put on a generic type, let me provide a short summary here, in code terms:

```
| type
  TSampleClass <T: class> = class
  TSampleRec <T: record> = class
  TCompClass <T: TButton> = class
  TInftClass <T: IGetValue> = class
  TConstrClass <T: constructor> = class
```

What you might not immediately realize after looking at constraints (and this certainly took me some time to get used to) is that you can combine them. For example, you can define a generic class limited to a sub-hierarchy and requiring also a given interface, like in:

```
| type
  TInftComp <T: TComponent, IGetValue> = class
```

Not all combinations make sense: for example you cannot specify both a class and a record, while using a class constraint combined with a specific class constraint would be redundant. Finally, notice that there is nothing like a method constraint, something that can be achieved with a single-method interface constraint (much more complex to express, though).

Predefined Generic Containers

Since the early days of templates in the C++ Language, one of the most obvious uses of template classes has been the definition of template containers or lists, up to the point that the C++ language defined a Standard Template Library (or STL).

When you define a list of objects, like Object Pascal's own `TObjectList`, you have a list that can potentially hold objects of any kind. Using either inheritance or composition you can indeed define custom containers for specific a type, but this is a tedious (and potentially error-prone) approach.

Object Pascal compilers come with a small set of generic container classes you can find in the `Generics.Collections` unit. The four core container classes are all implemented in an independent way (there is no inheritance among these classes), all implemented in a similar fashion (using a dynamic array), and are all mapped to the corresponding non-generic container class of the older `Containers` unit:

```
type
  TList<T> = class
  TQueue<T> = class
  TStack<T> = class
  TDictionary<TKey, TValue> = class
  TObjectList<T: class> = class(TList<T>)
  TObjectQueue<T: class> = class(TQueue<T>)
  TObjectStack<T: class> = class(TStack<T>)
  TObjectDictionary<TKey, TValue> = class(TDictionary<TKey, TValue>)
```

The logical difference among these classes should be quite obvious considering their names. A good way to test them, is to figure out how many changes you have to perform on existing code that uses a non-generic container class.

note The program uses only a few methods, so it is not a great test for interface compatibility between generic and non-generic lists, but I decided to take an existing program rather than fabricating one. Another reason for showing this demo, is that you might also have existing programs that don't use generic collection classes and will be encouraged to enhance them by taking advantage of this language feature.

Using TList<T>

The program, called `ListDemoMd2005`, has a unit defining a `TDate` class, and the main form used to refer to a `TList` of dates. As a starting point, I added a `uses` clause referring to `Generics.Collections`, then I changed the declaration of the main form field to:

422 - 14: Generics

```
private  
  FListDate: TList <TDate>;
```

Of course, the main form onCreate event handler that does create the list needed to be updated as well, becoming:

```
procedure TForm1.FormCreate(Sender: TObject);  
begin  
  FListDate := TList<TDate>.Create;  
end;
```

Now we can try to compile the rest of the code as it is. The program has a “wanted” bug, trying to add a TButton object to the list. The corresponding code used to compile and now fails:

```
procedure TForm1.ButtonWrongClick(Sender: TObject);  
begin  
  // add a button to the list  
  FListDate.Add (Sender); // Error:  
  // E2010 Incompatible types: 'TDate' and 'TObject'  
end;
```

The new list of dates is more robust in terms of type-checking than the original generic list pointers. Having removed that line, the program compiles and works. Still, it can be improved.

This is the original code used to display all of the dates of the list in a ListBox control:

```
var  
  I: Integer;  
begin  
  ListBox1.Clear;  
  for I := 0 to FListDate.Count - 1 do  
    ListBox1.Items.Add (  
      (TObject(FListDate [I]) as TDate).Text);
```

Notice the type cast, due to the fact that the program was using a list of pointers (TList), and not a list of objects (TObjectList). We can easily improve the program by writing:

```
  for I := 0 to FListDate.Count - 1 do  
    ListBox1.Items.Add (FListDate [I].Text);
```

Another improvement to this snippet can come from using an enumeration (something the predefined generic lists fully support) rather than a plain for loop:

```
var  
  ADate: TDate;  
begin  
  for ADate in FListDate do  
    begin  
      ListBox1.Items.Add (ADate.Text);  
    end;
```

Finally, the program can be improved by using a generic `TObjectList` owning the `TDate` objects, but that's a topic for the next section.

As I mentioned earlier, the `TList<T>` generic class has a high degree of compatibility. It has all the classic methods, like `Add`, `Insert`, `Remove`, and `IndexOf`. The `Capacity` and `Count` properties are there as well. Oddly, `Items` becomes `Item`, but being the default property (accessed by using the square brackets without the property name) you seldom explicitly refer to it anyway.

Sorting a `TList<T>`

What is interesting to understand is how `TList<T>` sorting works (my goal here is to add sorting support to the `ListDemoMd2005` application project). The `Sort` method is defined as:

```
procedure Sort; overload;  
procedure Sort(const AComparer: IComparer<T>); overload;
```

where the `IComparer<T>` interface is declared in the `Generics.Defaults` unit. If you call the first version the program, it will use the default comparer, initialized by the default constructor of `TList<T>`. In our case this will be useless.

What we need to do, instead, is to define a proper implementation of the `IComparer<T>` interface. For type compatibility, we need to define an implementation that works on the specific `TDate` class.

There are multiple ways to accomplish this, including using anonymous methods (covered in the next section even though that's a topic introduced in the next chapter). An interesting technique, also because it gives me the opportunity to show several usage patterns of generics, is to take advantage of a *structural* class that is part of the unit `Generics.Defaults` and is called `TComparer`.

note I'm calling this class *structural* because it helps defining the structure of the code, its architecture, but doesn't add a lot in terms of actual implementation. There might be a better name to refer to such a class, though.

The class is defined as an abstract and generic implementation of the interface, as follows:

```
type  
  TComparer<T> = class(TInterfacedObject, IComparer<T>)  
  public  
    class function Default: IComparer<T>;  
    class function Construct(  
      const Comparison: TComparison<T>): IComparer<T>;
```


424 - 14: Generics

```
function Compare(  
    const Left, Right: T): Integer; virtual; abstract;  
end;
```

What we have to do is instantiate this generic class for the specific data type (TDate, in the example) and also inherit a concrete class that implements the Compare method for the specific type. The two operations can be done at once, using a coding idiom that may take a while to digest:

```
type  
    TDateComparer = class (TComparer<TDate>)  
        function Compare(  
            const Left, Right: TDate): Integer; override;  
        end;
```

If you think this code looks very unusual, you're not alone. The new class inherits from a specific instance of the generic class, something you could express in two separate steps as:

```
type  
    TAnyDateComparer = TComparer<TDate>;  
    TMyDateComparer = class (TAnyDateComparer)  
        function Compare(  
            const Left, Right: TDate): Integer; override;  
        end;
```

note Having the two separate declarations might help reduce the generated code where you are reusing the base TAnyDateComparer type in the same unit.

You can find the actual implementation of the Compare function in the source code, as that's not the key point I want to stress here. Keep in mind, though, that even if you sort the list its IndexOf method won't take advantage of it (unlike the TStringList class).

Sorting with an Anonymous Method

The sorting code presented in the previous section looks quite complicated and it really is. It would be much easier and cleaner to pass the sorting function to the Sort method directly. In the past this was generally achieved by passing a function pointer. In Object Pascal this can be done by passing an anonymous method (a kind of method pointer, with several extra features, covered in detail in the next chapter).

note I suggest you have a look at this section even if you don't know much about anonymous methods, and then read it again after going through the next chapter.

The `IComparer<T>` parameter of the `Sort` method of the `TList<T>` class, in fact, can be used by calling the `Construct` method of `TComparer<T>`, passing an anonymous method as a parameter defined as:

```
type
  TComparison<T> = reference to function(
    const Left, Right: T): Integer;
```

In practice you can write a type-compatible function and pass it as parameter:

```
function DoCompare (const Left, Right: TDate): Integer;
var
  LDate, RDate: TDateTime;
begin
  LDate := EncodeDate(Left.Year, Left.Month, Left.Day);
  RDate := EncodeDate(Right.Year, Right.Month, Right.Day);
  if LDate = RDate then
    Result := 0
  else if LDate < RDate then
    Result := -1
  else
    Result := 1;
end;

procedure TForm1.ButtonAnonSortClick(Sender: TObject);
begin
  FListDate.Sort (TComparer<TDate>.Construct (DoCompare));
end;
```

note The `DoCompare` method above works like an anonymous method even if it does have a name. We'll see in a later code snippet that this is not required, though. Have patience until the next chapter for more information about this Object Pascal language construct. Notice also that with a `TDate` record I could have defined less than and greater than operators, making this code simpler, but even with a class I could have placed the comparison code in a method of the class.

If this looks quite traditional, consider you could have avoided the declaration of a separate function and pass it (its source code) as parameter to the `Construct` method, as follows:

```
procedure TForm1.ButtonAnonSortClick(Sender: TObject);
begin
  ListDate.Sort (TComparer<TDate>.Construct (
    function (const Left, Right: TDate): Integer
    var
      LDate, RDate: TDateTime;
    begin
      LDate := EncodeDate(Left.Year, Left.Month, Left.Day);
      RDate := EncodeDate(Right.Year, Right.Month, Right.Day);
      if LDate = RDate then
        Result := 0
      else if LDate < RDate then
        Result := -1
```

426 - 14: Generics

```
        else  
            Result := 1;  
        end));  
    end;
```

This example should have whet your appetite for learning more about anonymous methods! For sure, this last version is much simpler to write than the original comparison covered in the previous section, although for many Object Pascal developers having a derived class might look cleaner and be easier to understand (the inherited version separates the logic better, making potential code reuse easier, but many times you won't make use of it anyway).

Object Containers

Beside the generic classes covered at the beginning of this section, there are also four inherited generic classes that are derived from the base classes defined in the `Generics.Collections` unit, mimicking existing classes of the `Containers` unit:

```
type  
    TObjectList<T>: class = class(TList<T>)  
    TObjectQueue<T>: class = class(TQueue<T>)  
    TObjectStack<T>: class = class(TStack<T>)
```

Compared to their base classes, there are two key differences. One is that these generic types can be used only for objects; the second is that they define a customized `Notification` method, that in the case when an object is removed from the list (beside optionally calling the `OnNotify` event handler), will `Free` the object.

In other words, the `TObjectList<T>` class behaves like its non-generic counterpart when the `OwnsObjects` property is set. If you are wondering why this is not an option any more, consider that `TList<T>` can now be used directly to work with object types, unlike its non-generic counterpart.

There is also a fourth class, again, called `TObjectDictionary<TKey, TValue>`, which is defined in a different way, as it can own the key object, the value objects, or both of them. See the `TDictionaryOwnerships` set and the class constructor for more details.

Using a Generic Dictionary

Of all the predefined generic container classes, the one probably worth more detailed study is the generic dictionary, `TObjectDictionary<TKey, TValue>`.

note *Dictionary* in this case means a collection of elements each with a (unique) key value referring to it. (It is also known as an associative array.) In a classic dictionary you have words acting as keys for their definitions, but in programming terms the key doesn't have to be a string (even if this is a rather frequent case).

Other classes are just as important, but they seem to be easier to use and understand. As an example of using a dictionary, I've written an application that fetches data from a database table, creates an object for each record, and uses a composite index with a customer ID and a description as key. The reason for this separation is that a similar architecture can easily be used to create a proxy, in which the key takes the place of a light version of the actual object *loaded* from the database.

These are the two classes used by the CustomerDictionary application project for the key and the actual value. The first has only two relevant fields of the corresponding database table, while the second has the complete data structure (I've omitted the private fields, getter methods, and setter methods):

```
type
  TCustomerKey = class
  private
    ..
  published
    property CustNo: Double
      read FCustNo write SetCustNo;
    property Company: string
      read FCompany write SetCompany;
  end;

  TCustomer = class
  private
    ..
    procedure Init;
    procedure EnforceInit;
  public
    constructor Create (aCustKey: TCustomerKey);
    property CustKey: TCustomerKey
      read FCustKey write SetCustKey;
  published
    property CustNo: Double
      read GetCustNo write SetCustNo;
    property Company: string
      read GetCompany write SetCompany;
    property Addr1: string
      read GetAddr1 write SetAddr1;
    property City: string
      read GetCity write SetCity;
    property State: string
      read GetState write SetState;
    property Zip: string
      read GetZip write SetZip;
```

428 - 14: Generics

```
    property Country: string
      read GetCountry write SetCountry;
    property Phone: string
      read GetPhone write SetPhone;
    property FAX: string
      read GetFAX write SetFAX;
    property Contact: string
      read GetContact write SetContact;
  class var
    RefDataSet: TDataSet;
  end;
```

While the first class is very simple (each object is initialized when it is created), the `TCustomer` class uses a *lazy initialization* (or *proxy*) model and keeps around a reference to the source database shared (`class var`) by all objects. When an object is created it is assigned a reference to the corresponding `TCustomerKey`, while a class data field refers to the source dataset. In each getter method, the class checks if the object has indeed been initialized before returning the data, as in the following case:

```
function TCustomer.GetCompany: string;
begin
  EnforceInit;
  Result := FCompany;
end;
```

The `EnforceInit` method checks a local flag, eventually calling `Init` to load data from the database to the in-memory object:

```
procedure TCustomer.EnforceInit;
begin
  if not FInitDone then
    Init;
end;

procedure TCustomer.Init;
begin
  RefDataSet.Locate('CustNo', CustKey.CustNo, []);

  // could also load each published field via RTTI
  FCustNo := RefDataSet.FieldByName ('CustNo').AsFloat;
  FCompany := RefDataSet.FieldByName ('Company').AsString;
  FCountry := RefDataSet.FieldByName ('Country').AsString;
  ...
  FInitDone := True;
end;
```

Given these two classes, I've added a special purpose dictionary to the application. This custom dictionary class inherits from a generic class instantiated with the proper types and adds to it a specific method:

```
type
  TCustomerDictionary = class (
    TObjectDictionary <TCustomerKey, TCustomer>)
```

```

public
procedure LoadFromDataSet (Dataset: TDataSet);
end;

```

The loading method populates the dictionary, copying data in memory for only the key objects:

```

procedure TCustomerDictionary.LoadFromDataSet(Dataset: TDataSet);
var
    CustKey: TCustomerKey;
begin
    TCustomer.RefDataSet := dataset;
    Dataset.First;
    while not Dataset.EOF do
        begin
            CustKey := TCustomerKey.Create;
            CustKey.CustNo := Dataset ['CustNo'];
            CustKey.Company := Dataset ['Company'];
            self.Add(custKey, TCustomer.Create (CustKey));
            Dataset.Next;
        end;
    end;

```

The demo program has a main form and a data module hosting a ClientDataSet component. The main form has a ListView control that is filled when a user presses the only button.

note You might want to replace the ClientDataSet component with a real dataset, expanding the example considerably in terms of usefulness, as you could run a query for the keys and a separate one for the actual data of each single TCustomer object. I have similar code, but adding it here would have distracted us too much from the goal of the example, which is experimenting with a generic dictionary class.

After loading the data in the dictionary, the BtnPopulateClick method uses an enumerator on the dictionary's keys:

```

procedure TFormCustomerDictionary.BtnPopulateClick(
    Sender: TObject);
var
    Custkey: TCustomerKey;
    ListItem: TListItem;
begin
    DataModule1.ClientDataSet1.Active := True;
    CustDict.LoadFromDataSet(DataModule1.ClientDataSet1);
    for Custkey in CustDict.Keys do
        begin
            ListItem := ListView1.Items.Add;
            ListItem.Caption := Custkey.Company;
            ListItem.SubItems.Add(FloatToStr (Custkey.CustNo));
            ListItem.Data := Custkey;
        end;
    end;

```

430 - 14: Generics

This fills the first two columns of the ListView control, with the data available in the key objects. Whenever a user selects an item of the ListView control, though, the program will fill a third column:

```
procedure TFormCustomerDictionary.ListView1SelectItem(  
    Sender: TObject; Item: TListItem; Selected: Boolean);  
var  
    ACustomer: TCustomer;  
begin  
    ACustomer := CustDict.Items [Item.data];  
    Item.SubItems.Add(  
        IfThen (  
            ACustomer.State <> '',  
            ACustomer.State + ', ' + ACustomer.Country,  
            ACustomer.Country));  
end;
```

The method above gets the object mapped to the given key, and uses its data. Behind the scenes, the first time a specific object is used, the property access method triggers the loading of the entire data for the TCustomer object.

Dictionaries vs. String Lists

Over the years many Object Pascal developers, myself included, have overused the TStringList class. Not only you can use it for a plain list of strings and for a list of name/value pairs, but you can also use it to have a list objects associated with strings and search these objects. Since the introduction of generics, it is much better to use them instead of this use a favorite tool as a swiss-army knife kind of approach.

Specific and focused container classes are a much better option. For example, a generic TDictionary with a string key and an object-value will generally be better than a string list on two counts: cleaner and safer code, as there will be fewer type casts involved, and faster execution, given that dictionaries use hash tables.

To demonstrate these differences I've written a rather simple application, called StringListVsDictionary. Its main form stores two identical lists, declared as:

```
private  
    FList: TStringList;  
    FDict: TDictionary<string, TMyObject>;
```

The two lists are filled with random but identical entries with a cycle that repeats this code:

```
    FList.AddObject (AName, AnObject);  
    FDict.Add (AName, AnObject);
```

Two buttons retrieve each element of the list and does a search by name on each of them. Both methods scan the string list for the values, but the first locates the objects in the string list, while the second uses the dictionary. Notice that in the first case you need an as cast to get back the given type, while the dictionary is tied to that class already. Here is the main loop of the two methods:

```

TheTotal := 0;
for I := 0 to sList.Count -1 do
begin
  aName := FList[I];
  // now search for it
  anIndex := FList.IndexOf (AName);
  // get the object
  AnObject := FList.Objects [AnIndex] as TMyObject;
  Inc (TheTotal, AnObject.Value);
end;

TheTotal := 0;
for I := 0 to FList.Count -1 do
begin
  AName := FList[I];
  // get the object
  AnObject := FDict.Items [AName];
  Inc (TheTotal, AnObject.Value);
end;

```

I don't want to access the strings in sequence, but figure out how much time it takes to search in the sorted string list (which does a binary search) compared to the hashed keys of the dictionary. Not surprisingly the dictionary is faster, here are numbers in milliseconds for a test:

```

Total: 99493811
StringList: 2839
Total: 99493811
Dictionary: 686

```

The result is the same, given the initial values were identical, but the time is quite different, with the dictionary taking about *one fourth of the time* for a million entries.

Generic Interfaces

In the section “Sorting a TList<T>” you might have noticed the rather strange use of a predefined interface, which had a generic declaration. It is worth looking into this technique in detail, as it opens up significant opportunities.

432 - 14: Generics

The first technical element to notice is that it is perfectly legal to define a generic interface, as I've done in the `GenericInterface` application project:

```
type
  IGetValue<T> = interface
    function GetValue: T;
    procedure SetValue (Value: T);
end;
```

note This is the generic version of the `IGetValue` interface of the `IntfConstraints` application project, covered in the earlier section “Interface Constraints” of this chapter. In that case the interface had an `Integer` value, now it has a generic one.

Notice that differently from a standard interface, in case of a generic interface you don't need to specify a GUID to be used as Interface ID (or IID). The compiler will generate an IID for you for each instance of the generic interface, even if implicitly declared. In fact, you don't have to create a specific instance of the generic interface to implement it, but can define a generic class that implements the generic interface:

```
type
  TGetValue<T> = class (TInterfacedObject, IGetValue<T>)
  private
    FValue: T;
  public
    constructor Create (Value: T);
    destructor Destroy; override;
    function GetValue: T;
    procedure SetValue (Value: T);
end;
```

While the constructor assigns the initial value of the object, the destructor's only purpose is to log that an object was destroyed. We can create an instance of this generic class (thus generating a specific instance of the interface type behind the scenes) by writing:

```
procedure TFormGenericInterface.BtnValueClick(
  Sender: TObject);
var
  Aval: TGetValue<string>;
begin
  Aval := TGetValue<string>.Create (Caption);
  try
    Show ( 'TGetValue value: ' + Aval.GetValue);
  finally
    Aval.Free;
  end;
end;
```

An alternative approach, as we saw in the past for the `IntfConstraint` application project, is to use an interface variable of the corresponding type, making the specific interface type definition explicit (and not implicit as in the previous code snippet):

```
procedure TFormGenericInterface.BtnIValueClick(
    Sender: TObject);
var
    AVal: IGetValue<string>;
begin
    AVal := TGetValue<string>.Create (Caption);
    Show ('IGetValue value: ' + AVal.GetValue);
    // freed automatically, as it is reference counted
end;
```

Of course, we can also define a specific class that implements the generic interface, as in the following scenario (from the `GenericInterface` application project):

```
type
    TButtonValue = class (TButton, IGetValue<Integer>)
    public
        function GetValue: Integer;
        procedure SetValue (Value: Integer);
        class function MakeTButtonValue (Owner: TComponent;
            Parent: TwinControl): TButtonValue;
    end;
```

Notice that while the `TGetValue<T>` generic class implements the generic `IGetValue<T>` interface, the `TButtonValue` specific class implements the `IGetValue<Integer>` specific interface. Specifically, as in a previous example, the interface is remapped to the `Left` property of the control:

```
function TButtonValue.GetValue: Integer;
begin
    Result := Left;
end;
```

In the class above, the `MakeTButtonValue` class function is a ready-to-use method to create an object of the class. This method is used by the third button of the main form, as follows:

```
procedure TFormGenericInterface.BtnValueButtonClick(
    Sender: TObject);
var
    IVal: IGetValue<Integer>;
begin
    IVal := TButtonValue.MakeTButtonValue (self, ScrollBox1);
    Show ('Button value: ' + IntToStr (IVal.GetValue));
end;
```

Although it is totally unrelated to generic classes, here is the implementation of the `MakeTButtonValue` class function:

```
class function TButtonValue.MakeTButtonValue(
    Owner: TComponent; Parent: TwinControl): TButtonValue;
```

```

begin
  Result := TButtonValue.Create(Owner);
  Result.Parent := Parent;
  Result.SetBounds(Random (Parent.Width),
    Random (Parent.Height), Result.Width, Result.Height);
  Result.Text := 'Btnv';
end;

```

Predefined Generic Interfaces

Now that we have explored how to define generic interfaces and combine them with the use of generic and specific classes, we can get back to having a second look at the `Generics.Defaults` unit. This unit defines two generic comparison interfaces:

- `IComparer<T>` has a `Compare` method
- `IEqualityComparer<T>` has `Equals` and `GetHashCode` methods

These classes are implemented by some generic and specific classes, listed below (with no implementation details):

```

type
  TComparer<T> = class(TInterfacedObject, IComparer<T>)
  TEqualityComparer<T> = class(
    TInterfacedObject, IEqualityComparer<T>)
  TCustomComparer<T> = class(TSingletonImplementation,
    IComparer<T>, IEqualityComparer<T>)
  TStringComparer = class(TCustomComparer<string>)

```

In the listing above you can see that the base class used by the generic implementations of the interfaces is either the classic reference-counted `TInterfacedObject` class or the new `TSingletonImplementation` class. This is an oddly named class that provides a basic implementation of `IInterface` with no reference counting.

note The term singleton is generally used to define a class of which you can create only one instance, and not one with no reference counting. I consider this quite a misnomer.

As we have already seen in the “Sorting a `TList<T>`” section earlier in this chapter, these comparison classes are used by the generic containers. To make things more complicated, though, the `Generics.Default` unit relies quite heavily on anonymous methods, so you should probably look at it only after reading the next chapter.

Smart Pointers in Object Pascal

When approaching generics, you might get the wrong first impression that this language construct is mostly used for collections. While this is the simplest case for using generic classes, and very often the first example in books and docs, generics are useful well beyond the realm of collection (or container) classes. In the last example of this chapter I'm going to show you a *non-collection* generic type, that is the definition of a smart pointer.

If you come from an Object Pascal background, you might not have heard of smart pointers, an idea that comes from the C++ language. In C++ you can have pointers to objects, for which you have to manage memory directly and manually, and local object variables that are managed automatically but have many other limitations (including the lack of polymorphism). The idea of a smart pointer is to use a locally managed object to take care of the lifetime of the pointer to the real object you want to use. If this sounds too complicated, I hope the Object Pascal version (and its code) will help clarify it.

note The term polymorphisms in OOP languages is used to denote the situation in which you assign to a variable of a base class an object of a derived class and call one of the base class virtual methods, potentially ending up calling the version of the virtual method of the specific subclass.

Using Records for Smart Pointers

In Object Pascal objects are managed by reference, while records have a lifetime bound to the method in which they are declared. When the method ends, the memory area for the record is cleaned up. So what we can do is to use a record to manage the lifetime of an Object Pascal object.

Before Delphi 10.4, though, Object Pascal records offered no way to execute custom code at destruction time, a feature introduced with managed records. The old style mechanism, instead, was to use an interface field in the records, as that interface field is managed and the object used to implement the interface has its reference count decreased.

Another consideration is whether we want to use a standard record or a generic one. With a standard record having a TObject type field, you can delete that object when required, so that is sufficient in general terms. With a generic version, though, you can gain two advantages:

436 - 14: Generics

- A generic smart pointer can return a reference to the object it contains, so that you don't need to keep both references around
- A generic smart pointer can automatically create the container object, using a parameter-less constructor.

Here I'm only going to cover two examples of smart pointers implemented using generic records, even if this adds a little extra complexity. The starting point is going to be a generic record with a constraint to objects, like:

```
type
  TSmartPointer<T: class> = record
    strict private
      FValue: T;
      function GetValue: T;
    public
      constructor Create(AValue: T);
      property Value: T read GetValue;
    end;
```

The `Create` and `GetValue` methods of the record simply assign and read back the value. The use case scenario is the following code snippet which creates an object, creates a smart pointer wrapping it, and also allows to use the smart pointer to refer to the embedded object and call its methods (see the last line of code below):

```
var
  SL: TStringList;
begin
  SL := TStringList.Create;
  var SmartP: TSmartPointer<TStringList>.Create (SL);
  SL.Add( 'foo' );
  SmartP.Value.Add ( 'bar' );
```

As you may have worked out, this code causes a memory leak in the exact same way as without the smart pointer! In fact the record is destroyed as it goes out of scope, but it doesn't free the internal object.

Implementing a Smart Pointer with a Generic Managed Record

While the most relevant operation of the smart pointer record is its finalization, in the code below as you can see I'm adding also an initialization operator to set the object reference to `nil`. Ideally, we'd prevent any assignment operation (as having multiple link to the internal objects would require to configure a rather complex reference count mechanism), but given this isn't possible I added the operator and implemented it to raise an exception in case it is triggered.

This is the complete code of the generic managed record:

```
type
  TSmartPointer<T: class, constructor> = record
  strict private
    FValue: T;
    function GetValue: T;
  public
    class operator Initialize(out ARec: TSmartPointer <T>);
    class operator Finalize(var ARec: TSmartPointer <T>);
    class operator Assign(var ADest: TSmartPointer <T>;
      const [ref] ASrc: TSmartPointer <T>);
    constructor Create (AValue: T);
    property Value: T read GetValue;
end;
```

Notice that beside the class constraint the generic record has also a constructor constraint, because I want to be able to create objects of the generic data type. This happens in case the `GetValue` method is called and the field hasn't been initialized yet. This is the complete code of all of the methods:

```
constructor TSmartPointer<T>.Create(AValue: T);
begin
  FValue := AValue;
end;

class operator TSmartPointer<T>.Initialize(
  out ARec: TSmartPointer <T>);
begin
  ARec.FValue := nil;
end;

class operator TSmartPointer<T>.Finalize(
  var ARec: TSmartPointer<T>);
begin
  ARec.FValue.Free;
end;

class operator TSmartPointer<T>.Assign(
  var ADest: TSmartPointer <T>;
  const [ref] ASrc: TSmartPointer <T>);
begin
  raise Exception.Create(
    'Cannot copy or assign a TSmartPointer<T>');
end;

function TSmartPointer<T>.GetValue: T;
begin
  if not Assigned(FValue) then
    FValue := T.Create;
    Result := FValue;
end;
```

438 - 14: Generics

This code is part of the `SmartPointersSMR` project, which includes also example of how the smart pointer can be used. The first strictly resembles the sample code we considered using a few pages back:

```
procedure TFormSmartPointers.BtnSmartClick(Sender: TObject);  
var  
    SL: TStringList;  
begin  
    SL := TStringList.Create;  
    var SmartP := TSmartPointer<TStringList>.Create (SL);  
    SL.Add('foo');  
    SmartP.Value.Add('bar');  
    Log ('Count: ' + SL.Count.ToString);  
end;
```

However, given the generic smart pointer has support for automatic construction of an object of the indicated type, you can also get away with the explicit variable referring to the string list and the code to create it:

```
procedure TFormSmartPointers.BtnSmartShortClick(Sender: TObject);  
var  
    SmartP: TSmartPointer<TStringList>;  
begin  
    SmartP.Value.Add('foo');  
    SmartP.Value.Add('bar');  
    Log ('Count: ' + SmartP.Value.Count.ToString);  
end;
```

In the program, you can verify that all objects are actually destroyed and there is no memory leak by setting the global `ReportMemoryLeaksOnShutdown` to `True` in the initialization code. As a counter test, there is a button in the program that causes a leak, which is caught as the program terminates.

Implementing a Smart Pointer with a Generic Record and an Interface

As I already mentioned, before Delphi 10.4 made available managed records, a possible way to implement smart pointers was the use of an interface, given that the record will automatically free an object referenced by an interface field. While this approach is now less interesting, it does offer a couple of additional features like implicit conversion operators. Given this remains an interesting, complex example, I've decided to keep it, with a somehow reduced description (this is the `SmartPointers` project in the source).

To implement a smart pointer with an interface, you can write an internal, support class, tied to an interface, and use the interface reference counting mechanism to determine when to free the object. The internal class looks like the following:

```

type
  TFreeTheValue = class (TInterfacedObject)
  private
    FObjectToFree: TObject;
  public
    constructor Create(AnObjectToFree: TObject);
    destructor Destroy; override;
  end;

constructor TFreeTheValue.Create(
  AnObjectToFree: TObject);
begin
  FObjectToFree := AnObjectToFree;
end;

destructor TFreeTheValue.Destroy;
begin
  FObjectToFree.Free;
  inherited;
end;

```

I've declared this as a nested type of the generic smart pointer type. All we have to do in the smart pointer generic type, to enable this feature, is to add an interface reference and initialize it with a TFreeTheValue object referring to the contained object:

```

type
  TSmartPointer<T: class> = record
  strict private
    FValue: T;
    FFreeTheValue: IInterface;
    function GetValue: T;
  public
    constructor Create(AValue: T); overload;
    property Value: T read GetValue;
  end;

```

The pseudo-constructor becomes:

```

constructor TSmartPointer<T>.Create(AValue: T);
begin
  FValue := AValue;
  FFreeTheValue := TFreeTheValue.Create(FValue);
end;

```

With this code in place, we can now write the following code in a program without causing a memory leak (again the code is similar to what I listed initially and used in the manager record version):

440 - 14: Generics

```
procedure TFormSmartPointers.BtnSmartClick(  
    Sender: TObject);  
var  
    SL: TStringList;  
    SmartP: TSmartPointer<TStringList>;  
begin  
    SL := TStringList.Create;  
    SmartP.Create (SL);  
    SL.Add( 'foo' );  
    Show ( 'Count: ' + IntToStr (SL.Count));  
end;
```

At the end of the method the `SmartP` record is disposed, which causes its internal interfaced object to be destroyed, freeing the `TStringList` object.

note The code works even if an exception is raised. In fact, implicit `try-finally` blocks are being added all over the places by the compiler when you use a managed type, like in this case the record with an interface field.

Adding Implicit Conversion

With the managed records solution, we need to take some extra care in avoiding record copy operations, as this would require adding a manual reference counting mechanism and making the structure way more complex. However, given this is built into the interface-based solution, we can leverage this model to add conversion operators, which can simplify the initialization and creation of the data structure. Specifically, I'm going to add an `Implicit` conversion operator to *assign* the target object to the smart pointer:

```
class operator TSmartPointer<T>.  
    Implicit(AValue: T): TSmartPointer<T>;  
begin  
    Result := TSmartPointer<T>.Create(AValue);  
end;
```

With this code (and taking advantage of the `value` field) we can now write a more compact version of the code, like:

```
var  
    SmartP: TSmartPointer<TStringList>;  
begin  
    SmartP := TStringList.Create;  
    SmartP.Value.Add( 'foo' );  
    Show ( 'Count: ' + IntToStr (SmartP.Value.Count));
```

As an alternative, we can use a `TStringList` variable and use a more complicated constructor to initialize the smart pointer record even without an explicit reference to it:

```

var
  SL: TStringList;
begin
  SL := TSmartPointer<TStringList>.
    Create(TStringList.Create).Value;
  SL.Add('foo');
  Show('Count: ' + IntToStr(SL.Count));

```

As we've started down this road, we can also define the opposite conversion, and use the cast notation rather than the value property:

```

class operator TSmartPointer<T>.
  Implicit(AValue: T): TSmartPointer<T>;
begin
  Result := TSmartPointer<T>.Create(AValue);
end;

var
  SmartP: TSmartPointer<TStringList>;
begin
  SmartP := TStringList.Create;
  TStringList(SmartP).Add('foo2');

```

Now, you might also notice that I've always used a pseudo-constructor in the code above, but this is not needed on a record. All we need is a way to initialize the internal object, possibly calling its constructor, the first time we use it.

We cannot test if the internal object is `Assigned`, because records (unlike classes) are not initialized to zero. However we can perform that test on the interface variable, which is initialized.

Comparing Smart Pointer Solutions

The managed record version of the smart pointer is simpler and fairly effective, however the interface based version offers the advantage of conversion operators. They both have merit, although I personally tend to prefer the managed record version.

For more articulated analysis and more sophisticated solutions (out of the scope of this book) I can recommend the following blog post by Erik van Bilsen:

```

https://blog.grijjy.com/2020/08/12/custom-managed-records-for-smart-pointers/

```

Covariant Return Types with Generics

In general in Object Pascal (and most other static object-oriented languages) a method can return an object of a class but you cannot override it in a derived class to return a derived class object. This is a rather common practice called “Covariant Return Type” and explicitly supported by some languages like C++.

Of Animals, Dogs, and Cats

In coding terms, if `TDog` inherits from `TAnimal`, I'd want to have the methods:

```
function TAnimal.Get (AName: string): TAnimal;
function TDog.Get (AName: string): TDog;
```

However, in Object Pascal you cannot have virtual functions with a different return value, nor you can overload on the return type, but only when using different parameters. Let me show you the complete code of a simple demo. Here are the three classes involved:

```
type
  TAnimal = class
    private
      FName: string;
      procedure SetName(const Value: string);
    public
      property Name: string read FName write SetName;
    public
      class function Get (const AName: string): TAnimal;
      function ToString: string; override;
    end;

  TDog = class (TAnimal)
  end;

  TCat = class (TAnimal)
  end;
```

The implementation of the two methods is quite simple, once you notice that the class function is actually used to create new objects, internally calling a constructor. The reason is I don't want to create a constructor directly is that this is a more general technique, in which a method of a class can create objects of other classed (or class hierarchies). This is the code:

```
class function TAnimal.Get(const AName: string): TAnimal;
begin
```

```

    Result := Create;
    Result.FName := AName;
end;

function TAnimal.ToString: string;
begin
    Result := 'This ' + Copy (ClassName, 2, MaxInt) +
        ' is called ' + FName;
end;

```

Now we can use the class by writing the following code, which is what I don't terribly like, given we have to cast back the result to the proper type:

```

var
    ACat: TCat;
begin
    ACat := TCat.Get('Matisse') as TCat;
    Memo1.Lines.Add (ACat.ToString);
    ACat.Free;

```

Again, what I'd like to do is to be able to assigned the value returned by `TCat.Get` to a reference of the `TCat` class without an explicit cast. How can we do that?

A Method with a Generic Result

It turns out generics can help us solve the problem. Not generic types, which is the most commonly used form of generics. But generic methods for non-generic types, discussed earlier in this chapter. What I can add to the `TAnimal` class is a method with a *generic type* parameter, like:

```

class function GetAs<T: class> (const AName: string): T;

```

This method requires a generic type parameter, which needs to be a class (or instance type) and returns an object of that type. A sample implementation is here:

```

class function TAnimal.GetAs<T>(const AName: string): T;
var
    Res: TAnimal;
begin
    Res := Get (aName);
    if res.inheritsFrom (T) then
        Result := T(Res)
    else
        Result := nil;
end;

```

Now we can create an instance and using it omitting the *as* cast, although we still have to pass the type as parameter:

```

var
    ADog: TDog;
begin

```

```
ADog := TDog.GetAs<TDog>('Pluto');
Memo1.Lines.Add (ADog.ToString);
ADog.Free;
```

Returning a Derived Object of a Different Class

When you return an object of the same class, you can replace this code with a proper use of constructors. But the use of generics to obtain covariant return types is actually more flexible. In fact we can use it to return objects of a different class, or hierarchy of classes:

```
type
  TAnimalShop = class
    class function GetAs<T: TAnimal, constructor> (
      const AName: string): T;
  end;
```

note A class like this, used to create objects of a different class (or more than one class, depending on the parameters) is generally called a “*class factory*”.

We can now use the specific class constraint (something impossible in the class itself) and we have to specify the constructor constraint to be able to create an object of the given class from within the generic method:

```
class function TAnimalShop.GetAs<T>(const AName: string): T;
var
  Res: TAnimal;
begin
  Res := T.Create;
  Res.Name := AName;
  if res.inheritsFrom (T) then
    Result := T(Res)
  else
    Result := nil;
end;
```

Notice that now in the call we don't have to repeat the class type twice:

```
ADog := TAnimalShop.GetAs<TDog>('Pluto');
```

15: anonymous methods

The Object Pascal language includes both procedural types (that is, types declaring pointers to procedures and functions) and method pointers (that is, types declaring pointers to methods).

note In case you want more information, procedural types were covered in Chapter 4 while events and method pointer types were described Chapter 10.

Although you might seldom use them directly, these are key features of Object Pascal that every developer works with. In fact, method pointers types are the foundation for event handlers in components and visual controls: every time you declare an event handler, even a simple `OnClick` you are in fact declaring a method that will be connected to an event (the `OnClick` event, in this case) using a method pointer.

Anonymous methods extend this feature by letting you pass the actual code of a method as a parameter, rather than the name of a method defined elsewhere. This is not the only difference, though. What makes anonymous methods very different from other techniques is the way they manage the lifetime of local variables.

446 - 15: Anonymous Methods

The definition above matches with a feature called closures in many other languages, for example JavaScript. If Object Pascal anonymous methods are in fact closures, how come the language refers to them using a different term? The reason lies in the fact both terms are used by different programming languages and that the C++ compiler produced by Embarcadero uses the term closures for what Object Pascal calls event handlers.

Anonymous methods have been around in different forms and with different names for many years in quite a few programming languages, most notably dynamic languages. I've had extensive experience with closures in JavaScript, particularly with the jQuery library and AJAX calls. The corresponding feature in C# is called an anonymous delegate.

But here I don't want to devote too much time comparing closures and related techniques in the various programming languages, but rather describe in detail how they work in Object Pascal.

note From a very high perspective, generics allows code to be code parametrized for a type, anonymous methods allows code to be parametrized for a method.

Syntax and Semantics of Anonymous Methods

An anonymous method in Object Pascal is a mechanism to *create a method value in an expression context*. A rather cryptic definition, but a rather precise one given it underlines the key difference from method pointers, the *expression context*. Before we get to this, though, let me start from the beginning with a very simple code example (included in the `AnonymFirst` application project along with most other code snippets in this section).

This is the declaration of an anonymous method type, something you need to provide given that Object Pascal is a strongly typed language:

```
type
  TIntProc = reference to procedure (N: Integer);
```

This is different from a method reference type only in the keywords being used for the declaration:

```
type
  TIntMethod = procedure (N: Integer) of object;
```

An Anonymous Method Variable

Once you have an anonymous method type you can, in the simplest cases, declare a variable of this type, assign a type-compatible anonymous method, and call the method through the variable:

```
procedure TFormAnonymFirst.BtnSimpleVarClick(Sender: TObject);
var
  AnIntProc: TIntProc;
begin
  AnIntProc :=
    procedure (N: Integer)
    begin
      Memo1.Lines.Add (IntToStr (N));
    end;
  AnIntProc (22);
end;
```

Notice the syntax used to assign an actual procedure, with in-place code, to the local variable `AnIntProc`.

An Anonymous Method Parameter

As a more interesting example (with even more surprising syntax), we can pass an anonymous method as parameter to a function. Suppose you have a function taking an anonymous method parameter:

```
procedure CallTwice (Value: Integer; AnIntProc: TIntProc);
begin
  AnIntProc (Value);
  Inc (Value);
  AnIntProc (Value);
end;
```

The function calls the method passed as parameter twice with two consecutive integers values, the one passed as parameter and the following one. You call the function by passing an actual anonymous method to it, with directly in-place code that looks surprising:

```
procedure TFormAnonymFirst.BtnProcParamClick(Sender: TObject);
begin
  CallTwice (48,
    procedure (N: Integer)
    begin
      Show (IntToHex (N, 4));
    end);
  CallTwice (100,
    procedure (N: Integer)
    begin
```


448 - 15: Anonymous Methods

```
        Show (FloatToStr(Sqrt(N)));  
    end);  
end;
```

From the syntax point of view notice the procedure passed as parameter within parentheses and not terminated by a semicolon. The actual effect of the code is to call the `IntToHex` with 48 and 49 and the `FloatToStr` on the square root of 100 and 101, producing the following output:

```
0030  
0031  
10  
10.0498756211209
```

Using Local Variables

We could have achieved the same effect using method pointers albeit with a different and less readable syntax. What makes anonymous method clearly different is the way they can refer to local variables of the calling method. Consider the following code:

```
procedure TFormAnonymFirst.BtnLocalValClick(Sender: TObject);  
var  
    ANumber: Integer;  
begin  
    ANumber := 0;  
    CallTwice (10,  
        procedure (N: Integer)  
        begin  
            Inc (ANumber, N);  
        end);  
    Show (IntToStr (ANumber));  
end;
```

Here the method, still passed to the `CallTwice` procedure, uses the local parameter `N`, but also a local variable from the context from which it was called, `ANumber`. What's the effect? The two calls of the anonymous method will modify the local variable, adding the parameter to it, 10 the first time and 11 the second. The final value of `ANumber` will be 21.

Extending the Lifetime of Local Variables

The previous example shows an interesting effect, but with a sequence of nested function calls, the fact you can use the local variable isn't that surprising. The power

of anonymous methods, however, lies in the fact they can use a local variable and also extend its lifetime until needed. An example will prove the point more than a lengthy explanation.

note In slightly more technical details, anonymous methods copy the variables and parameters they use to the heap when they are created, and keep them alive as long as the specific instance of the anonymous method.

I've added (using class completion) to the `TFormAnonymFirst` form class of the `AnonymFirst` application project a property of an anonymous method pointer type (well, actually the same anonymous method pointer type I've used in all of the code of the project):

```
private
  FAnonMeth: TIntProc;
  procedure SetAnonMeth(const Value: TIntProc);
public
  property AnonMeth: TIntProc
    read FAnonMeth write SetAnonMeth;
```

Then I've added two more buttons to the form of the program. The first saves the property an anonymous method that uses a local variable (more or less like in the previous `BtnLocalValClick` method):

```
procedure TFormAnonymFirst.BtnStoreClick(Sender: TObject);
var
  ANumber: Integer;
begin
  ANumber := 3;
  AnonMeth :=
    procedure (N: Integer)
    begin
      Inc (ANumber, N);
      Show (IntToStr (ANumber));
    end;
end;
```

When this method executes the anonymous method is not executed, only stored. The local variable `ANumber` is initialized to three, is not modified, goes out of local scope (as the method terminates), and is displaced. At least, that is what you'd expect from standard Object Pascal code.

The second button I added to the form for this specific step calls the anonymous method stored in the `AnonMeth` property:

```
procedure TFormAnonymFirst.BtnCallClick(Sender: TObject);
begin
  if Assigned (AnonMeth) then
  begin
    CallTwice (2, AnonMeth);
```

450 - 15: Anonymous Methods

```
end;  
end;
```

When this code is executed, it calls an anonymous method that uses the local variable `ANumber` of a method that's not on the stack any more. However, since anonymous methods *capture* their execution context the variable is still there and can be used as long as that given instance of the anonymous method (that is, a reference to the method) is around.

As a further proof, do the following. Press the `Store` button once, the `Call` button two times and you'll see that the same *captured* variable is being used:

```
5  
8  
10  
13
```

note The reason for this sequence is that the value starts at 3, each call to `CallTwice` passed its parameter to the anonymous methods a first time (that is 2) and then a second time after incrementing it (that is, the second time it passes 3).

Now press `Store` once more and press `Call` again. What happens, why is the value of the local variable reset? By assigning a new anonymous method instance, the old anonymous method is deleted (along with its own execution context) and a new execution context is capture, including a new instance of the local variable. The full sequence *Store – Call – Call – Store – Call* produces:

```
5  
8  
10  
13  
5  
8
```

It is the implication of this behavior, resembling what some other languages do, that makes anonymous methods an extremely powerful language feature, which you can use to implement something that simply wasn't possible in the past.

Anonymous Methods Behind the Scenes

If the variable capture feature is one of the most relevant for anonymous methods, there are a few more techniques that are worth looking at, before we focus on some

real world examples. *Still, if you are new to anonymous methods, you might want to skip this rather advanced section and come back during a second read.*

The (Potentially) Missing Parenthesis

Notice that in the code above I used the `AnonMeth` symbol to refer to the anonymous method, not to invoke it. For invoking it, I should have typed:

```
| AnonMeth (2)
```

The difference is clear; I need to pass a proper parameter to invoke the method. Things are slightly more confusing with parameterless anonymous methods. If you declare:

```
| type
    TAnyProc = reference to procedure;
var
    AnyProc: TAnyProc;
```

The call to `AnyProc` must be followed by the empty parentheses, otherwise the compiler thinks you are trying to get the method (its address) rather than call it:

```
| AnyProc ();
```

Something similar happens when you call a function that returns an anonymous method, as in the following case taken from the usual `AnonymFirst` application project:

```
| function GetShowMethod: TIntProc;
var
    X: Integer;
begin
    X := Random (100);
    ShowMessage ('New x is ' + IntToStr (X));
    Result :=
        procedure (N: Integer)
        begin
            X := X + N;
            ShowMessage (IntToStr (X));
        end;
end;
```

Now the question is, how do you call it? If you simply call

```
| GetShowMethod;
```

It compiles and executes, but all it does is call the anonymous method assignment code, throwing away the anonymous method returned by the function.

How do you call the actual anonymous method passing a parameter to it? One option is to use a temporary anonymous method variable:

452 - 15: Anonymous Methods

```
var  
  Ip: TIntProc;  
begin  
  Ip := GetShowMethod();  
  Ip (3);
```

Notice in this case the parentheses after the `GetShowMethod` call. If you omit them (a standard Pascal practice) you'll get the following error:

```
E2010 Incompatible types: 'TIntProc' and 'Procedure'
```

Without the parentheses the compiler thinks you want to assign the `GetShowMethod` function itself, and not its result to the `Ip` method pointer. Still, using a temporary variable might not be the best option in this case, as it makes the code unnaturally complex. A simple call

```
GetShowMethod(3);
```

won't compile, as you cannot pass a parameter to the method. You need to add the empty parenthesis to the first call, and the Integer parameter to the resulting anonymous method. Oddly enough, you can write:

```
GetShowMethod()(3);
```

Anonymous Methods Implementation

What happens behind the scenes in the implementation of anonymous methods? The actual code generated by the compiler for anonymous methods is based on interfaces, with a single (hidden) invocation method called `Invoke`, plus the usual reference counting support (that's useful to determine the lifetime of anonymous methods and the context they capture).

Getting details of the internals is probably very complicated and of limited worth. Suffice to say that the implementation is very efficient, in terms of speed, and requires about 500 extra bytes for each anonymous method.

In other words, a method reference in Object Pascal is implemented with a *special* single method interface, with a compiler-generated method having the same signature as the method reference it is implementing. The interface takes advantage of reference counting for its automatic disposal.

note Although practically the interface used for an anonymous method looks like any other interface, the compiler distinguishes between these *special* interfaces so you cannot mix them in code.

Beside this hidden interface, for each invocation of an anonymous method the compiler creates a hidden object that has the method implementation and the data

required to *capture* the invocation context. That's how you get a new set of captured variables for each call of the method.

Ready To Use Reference Types

Every time you use an anonymous method as a parameter you need to define a corresponding reference pointer data type. To avoid the proliferation of local types, Object Pascal provides a number of ready-to-use reference pointer types in the `System.SysUtils` unit. As you can see in the code snippet below, most of these type definitions use parametrized types, so that with a single generic declaration you have a different reference pointer type for each possible data type:

```
type
  TProc = reference to procedure;
  TProc<T> = reference to procedure (Arg1: T);
  TProc<T1,T2> = reference to procedure (
    Arg1: T1; Arg2: T2);
  TProc<T1,T2,T3> = reference to procedure (
    Arg1: T1; Arg2: T2; Arg3: T3);
  TProc<T1,T2,T3,T4> = reference to procedure (
    Arg1: T1; Arg2: T2; Arg3: T3; Arg4: T4);
```

Using these declarations, you can define procedures that take anonymous method parameters like in the following:

```
procedure UseCode (Proc: TProc);
function DoThis (Proc: TProc): string;
function DoThat (ProcInt: TProc<Integer>): string;
```

In the first and second case you pass a parameterless anonymous method, in the third you pass a method with a single Integer parameter:

```
UseCode (
  procedure
  begin
    ...
  end);
StrRes := DoThat (
  procedure (I: Integer)
  begin
    ...
  end);
```

Similarly the `System.SysUtils` unit defines a set of anonymous method types with a generic return value:

```
type
  TFunc<TResult> = reference to function: TResult;
  TFunc<T,TResult> = reference to function (
    Arg1: T): TResult;
```

454 - 15: Anonymous Methods

```
TFunc<T1,T2,TResult> = reference to function (  
    Arg1: T1; Arg2: T2): TResult;  
TFunc<T1,T2,T3,TResult> = reference to function (  
    Arg1: T1; Arg2: T2; Arg3: T3): TResult;  
TFunc<T1,T2,T3,T4,TResult> = reference to function (  
    Arg1: T1; Arg2: T2; Arg3: T3; Arg4: T4): TResult;  
TPredicate<T> = reference to function (  
    Arg1: T): Boolean;
```

These definitions are very broad, as you can use countless combinations of data types for up to four parameters and a return type. The last definition is very similar to the second, but corresponds to a specific case that is very frequent, a function taking a generic parameter and returning a Boolean.

Anonymous Methods in the Real World

At first sight, it is not easy to fully understand the power of anonymous methods and the scenarios that can benefit from using them. That's why rather than coming out with more convoluted examples covering the language, I decided to focus on some that have a practical impact and provide starting points for further exploration.

Anonymous Event Handlers

One of the distinguishing features of Object Pascal has been its implementation of event handlers using method pointers. Anonymous methods can be used to attach a new behavior to an event without having to declare a separate method and capturing the method's execution context. This avoids having to add extra fields to a form to pass parameters from one method to another.

As an example (the *AnonButton* application project), I've added an *anonymous click* event to a button, declaring a proper method pointer type and adding a new event handler to a custom button class (defined using an interposer class):

```
type  
    TAnonNotif = reference to procedure (Sender: TObject);  
  
    // interposer class  
    TButton = class (FMX.StdCtrls.TButton)  
    private
```

```

    FAnonClick: TAnonNotif;
    procedure SetAnonClick(const Value: TAnonNotif);
    public
        procedure Click; override;
    public
        property AnonClick: TAnonNotif
            read FAnonClick write SetAnonClick;
    end;

```

note An *interposer* class is a derived class having the same name as its base class. Having two classes with the same name is possible because the two classes are in different units, so their full name (*unitname.classname*) is different. Declaring an interposer class can be handy as you can simply place a Button control on the form and attach extra behavior to it, without having to install a new component in the IDE and replace the controls on your form with the new type. The only trick you have to remember is that if the definition of the interposer class is in a separate unit (not the form unit as in this simple example), that unit has to be listed in the uses statement after the unit defining the base class.

The code of this class is fairly simple, as the setter method saves the new pointer and the Click method calls it before doing the standard processing (that is, calling the onClick event handler if available):

```

    procedure TButton.SetAnonClick(const Value: TAnonNotif);
    begin
        FAnonClick := Value;
    end;

    procedure TButton.Click;
    begin
        if Assigned (FAnonClick) then
            FAnonClick (self)
        inherited;
    end;

```

How can you use this new event handler? Basically you can assign an anonymous method to it:

```

    procedure TFormAnonButton.BtnAssignClick(Sender: TObject);
    begin
        BtnInvoke.AnonClick :=
            procedure (Sender: TObject)
            begin
                Show ((Sender as TButton).Text);
            end;
    end;

```

Now this looks rather pointless, as the same effect could easily be achieved using a standard event handler method. The following, instead, starts making a difference, as the anonymous method captures a reference to the component that assigned the event handler, by referencing the Sender parameter.

456 - 15: Anonymous Methods

This can be done after temporarily assigning it to a local variable, as the `Sender` parameter of the anonymous method hides the `BtnKeepRefClick` method's `Sender` parameter:

```
procedure TFormAnonButton.BtnKeepRefClick(Sender: TObject);  
var  
    ACompRef: TComponent;  
begin  
    ACompRef := Sender as TComponent;  
    BtnInvoke.AnonClick :=  
        procedure (Sender: TObject)  
        begin  
            Show ((Sender as TButton).Text +  
                ' assigned by ' + ACompRef.Name);  
        end;  
end;
```

As you press the `BtnInvoke` button, you'll see its caption along with the name of the component that assigned the anonymous method handler.

Timing Anonymous Methods

Developers frequently add timing code to existing routines to compare their relative speed. Supposing you have two code fragments and you want to compare their speed by executing them a few million times, you could write the following which is taken from the `LargeString` application project of Chapter 6:

```
procedure TForm1.Button1Click(Sender: TObject);  
var  
    Str1, Str2: string;  
    I: Integer;  
    T1: TStopwatch;  
begin  
    Str1 := 'Marco '  
    Str2 := 'Cantu '  
  
    T1 := TStopwatch.StartNew;  
    for I := 1 to MaxLoop do  
        Str1 := Str1 + Str2;  
  
    T1.Stop;  
    Show('Length: ' + Str1.Length.ToString);  
    Show('Concatenation: ' + T1.ElapsedMilliseconds.ToString);  
end;
```

A second method has similar code but used the `TStringBuilder` class rather than plain concatenation. Now we can take advantage of anonymous methods to create a timing skeleton and pass the specific code as parameter, as I've done in an updated version of the code, in the `AnonLargeStrings` application project.

Rather than repeating the timing code over and over, you can write a function with the timing code that would invoke the code snippet through a parameterless anonymous method:

```
function TimeCode (NLoops: Integer; Proc: TProc): string;
var
    T1: TStopwatch;
    I: Integer;
begin
    T1 := TStopwatch.StartNew;
    for I := 1 to NLoops do
        Proc;
    T1.Stop;
    Result := T1.ElapsedMilliseconds.ToString;
end;

procedure TForm1.Button1Click(Sender: TObject);
var
    Str1, Str2: string;
begin
    Str1 := 'Marco ';
    Str2 := 'Cantu ';
    Show ('Concatenation: ' +
        TimeCode (MaxLoop,
            procedure ()
            begin
                Str1 := Str1 + Str2;
            end));
    Show('Length: ' + Str1.Length.ToString);
end;
```

Notice, though, that if you execute the standard version and the one based on anonymous methods you'll get a slightly different output, the anonymous method version sees a penalty of roughly 10%. The reason is that rather than directly executing the local code, the program has to make a virtual call to the anonymous method implementation. As this difference is consistent, the testing code makes perfect sense anyway.

However, if you need to squeeze performance from your code, using anonymous methods won't be as fast as directly writing the code, with using a direct function. Using a non-virtual method pointer would probably be somewhere in between the two in terms of performance.

Threads Synchronization

In multithreaded applications that need to update the user interface, you cannot access properties of visual components (or in memory-objects) that are part of the main thread without a synchronization mechanism. Delphi visual component

458 - 15: Anonymous Methods

libraries aren't inherently thread-safe (as is true for most user-interface libraries). Two threads accessing an object at the same time could compromise its state.

The classic solution offered by the `TThread` class in Object Pascal is to call a special method, `Synchronize`, passing as a parameter the reference to another method, the one to be executed safely. This second method cannot have parameters, so it is common practice to add extra fields to the thread class to pass the information from one method to another.

As a practical example, in the book *Mastering Delphi 2005* I wrote a `WebFind` application (a program that runs searches on Google via HTTP and extracts the resulting links from the HTML of the page), with the following thread class:

```
type
  TFindWebThread = class(TThread)
  protected
    FAddr, FText, FStatus: string;
    procedure Execute; override;
    procedure AddToList;
    procedure ShowStatus;
    procedure GrabHtml;
    procedure HtmlToList;
    procedure Httpwork (Sender: TObject;
      AWorkMode: TWorkMode; AWorkCount: Int64);
  public
    FStrUrl: string;
    FStrRead: string;
  end;
```

The three protected string fields and some of the extra methods have been introduced to support synchronization with the user interface. For example, the `Httpwork` event handler hooked to the `OnReceivedData` event of an internal `THTTPClient` component (a component part of the Delphi HTTP Client Library), used to have the following code, that called the `ShowStatus` method:

```
procedure TFindWebThread.Httpwork(const Sender: TObject;
  AContentLength, AReadCount: Int64; var AAbort: Boolean);
begin
  FStatus := 'Received ' + IntToStr (AReadCount) + ' for ' + FStrUrl;
  Synchronize (ShowStatus);
end;

procedure TFindWebThread.ShowStatus;
begin
  Form1.StatusBar1.SimpleText := FStatus;
end;
```

The `Synchronize` method of the Object Pascal RTL has two different overloaded definitions:

```
type
  TThreadMethod = procedure of object;
```

```

TThreadProcedure = reference to procedure;

TThread = class
...
procedure Synchronize(AMethod: TThreadMethod); overload;
procedure Synchronize(AThreadProc: TThreadProcedure); overload;

```

For this reason we can remove the FStatus text field and the ShowStatus function, and rewrite the Httpwork event handler using the new version of Synchronize and an anonymous method:

```

procedure TFindWebThreadAnon.Httpwork(const Sender: TObject;
  AContentLength, AReadCount: Int64; var AAbort: Boolean);
begin
  Synchronize (
    procedure
    begin
      Form1.StatusBar1.SimpleText :=
        'Received ' + IntToStr (AReadCount) + ' for ' + FStrUrl;
    end);
end;

```

Using the same approach throughout the code of the class, the thread class becomes the following (you can find both thread classes in the version of the webFind application project that comes with the source code of this book):

```

type
  TFindWebThreadAnon = class(TThread)
  protected
    procedure Execute; override;
    procedure GrabHtml;
    procedure HtmlToList;
    procedure Httpwork (const Sender: TObject; AContentLength: Int64;
      AReadCount: Int64; var AAbort: Boolean);
  public
    FStrUrl: string;
    FStrRead: string;
  end;

```

Using anonymous methods simplifies the code needed for thread synchronization, as you can avoid temporary fields.

note Anonymous methods have a lot of relationships with threading, because a thread is used to run some code and anonymous method represent code. This is why there is support in the TThread class to use them, but also in the Parallel Programming Library (in TParallel.For and to define a TTask). Given examining multithreading goes well beyond this chapter, I won't add more examples in this direction. Still, I'm going to use another thread in the next example, because this is most often a requirement when making an HTTP call.

AJAX in Object Pascal

The last example in this section, the `AnonAjax` application demo, is one of my favorite examples of anonymous methods (even if a bit extreme). The reason is that I learned using closures (or anonymous methods) in JavaScript, while writing AJAX applications with the jQuery library a few years back.

note The AJAX acronym stands for Asynchronous JavaScript XML, as this was originally the format used in web services calls done from the browser. As this technology became more popular and widespread, and web services moved to the REST architecture and the JSON format, the term AJAX has faded away a bit, in favor of REST. I've decided to keep this old name for the demo anyway, given it explains the purpose of the application, and the history behind it. You can read more at: [https://en.wikipedia.org/wiki/Ajax_\(programming\)](https://en.wikipedia.org/wiki/Ajax_(programming)).

The `AjaxCall` global function spawns a thread, passing to the thread an anonymous method to execute on completion. The function is just a wrapper around the thread constructor:

```
type
  TAjaxCallback = reference to procedure (
    ResponseContent: TStringStream);

procedure AjaxCall (const StrUrl: string;
  AjaxCallback: TAjaxCallback);
begin
  TAjaxThread.Create (StrUrl, AjaxCallback);
end;
```

All of the code is in the `TAjaxThread` class, a thread class with an internal HTTP client component (from the Delphi HTTP Client library) used to access a given URL, asynchronously:

```
type
  TAjaxThread = class (TThread)
  private
    FHttp: THTTPClient;
    FURL: string;
    FAjaxCallback: TAjaxCallback;
  protected
    procedure Execute; override;
  public
    constructor Create (const StrUrl: string;
      AjaxCallback: TAjaxCallback);
    destructor Destroy; override;
  end;
```

The constructor does some initialization, copying its parameters to the corresponding local fields of the thread class and creating the `FHttp` object. The real meat of the class is in its `Execute` method, which does the HTTP request, saving the result in a

stream that is later reset and passed to the callback function – the anonymous method:

```
procedure TAjaxThread.Execute;
var
  AResponseContent: TStringStream;
begin
  AResponseContent := TStringStream.Create;
  try
    FHttp.Get (FURL, AResponseContent);
    AResponseContent.Position := 0;
    FAjaxCallback (AResponseContent);
  finally
    AResponseContent.Free;
  end;
end;
```

As an example of its usage, the AnonAjax application project has a button to copy the content of a Web page to a Memo control (after adding the requested URL):

```
procedure TFormAnonAjax.BtnReadClick(Sender: TObject);
begin
  AjaxCall (EdUrl.Text,
    procedure (AResponseContent: TStringStream)
    begin
      Memo1.Lines.Text := AResponseContent.DataString;
      Memo1.Lines.Insert (0, 'From URL: ' + EdUrl.Text);
    end);
end;
```

After the HTTP request has finished, you can do any processing on it. An example would be to extract links from the HTML file (in a way that resembles the `webFind` application covered earlier). Again, to make this function flexible, it takes as a parameter the anonymous method to execute for each link:

```
type
  TLinkCallback = reference to procedure (const StrLink: string);

procedure ExtractLinks (StrData: string; ProcLink: TLinkCallback);
var
  strAddr: string;
  NBegin, NEnd: Integer;
begin
  StrData := LowerCase (StrData);
  NBegin := 1;
  repeat
    nBegin := PosEx ('href="http', StrData, NBegin);
    if NBegin <> 0 then
      begin
        // find the end of the HTTP reference
        NBegin := NBegin + 6;
        NEnd := PosEx ('"', StrData, NBegin);
        StrAddr := Copy (StrData, NBegin, NEnd - NBegin);
        // move on
```

462 - 15: Anonymous Methods

```
        NBegin := NEnd + 1;  
        // execute anon method  
        ProcLink (StrAddr)  
    end;  
    until NBegin = 0;  
end;
```

If you apply this function to the result of an AJAX call and provide a further method for processing, you end up with two nested anonymous method calls, like in the second button of the AnonAjax application project:

```
procedure TFormAnonAjax.BtnLinksClick(Sender: TObject);  
begin  
    AjaxCall (EdUrl.Text,  
        procedure (AResponseContent: TStringStream)  
        begin  
            ExtractLinks(AResponseContent.DataString,  
                procedure (const AUrl: string)  
                begin  
                    Memo1.Lines.Add (AUrl + ' in ' + EdUrl.Text);  
                end);  
        end);  
end;
```

In this case the Memo control will receive a collection of links, instead of the HTML of the returned page. A variation to the link extraction routine above would be an image extraction routine. The `ExtractImages` function grabs the source (`src`) of the `img` tags of the HTML file returned, and calls another `TLinkCallback`-compatible anonymous method (see the source code for the function details).

Now you can envision opening an HTML page (with the `AjaxCall` function), extract the image links, and use `AjaxCall` again to grab the actual images. This means using a triple-nested closure, in a coding structure that some Object Pascal programmers might find very unusual, but that is certainly very powerful and expressive:

```
procedure TFormAnonAjax.BtnImagesClick(Sender: TObject);  
var  
    NHit: Integer;  
begin  
    NHit := 0;  
    AjaxCall (EdUrl.Text,  
        procedure (AResponseContent: TStringStream)  
        begin  
            ExtractImages(AResponseContent.DataString,  
                procedure (const AUrl: string)  
                begin  
                    Inc (NHit);  
                    Memo1.Lines.Add (IntToStr (NHit) + '.' +  
                        AUrl + ' in ' + EdUrl.Text);  
                    if NHit = 1 then // load the first  
                    begin  
                        var RealURL := IfThen (AURL[1]='/',
```

```

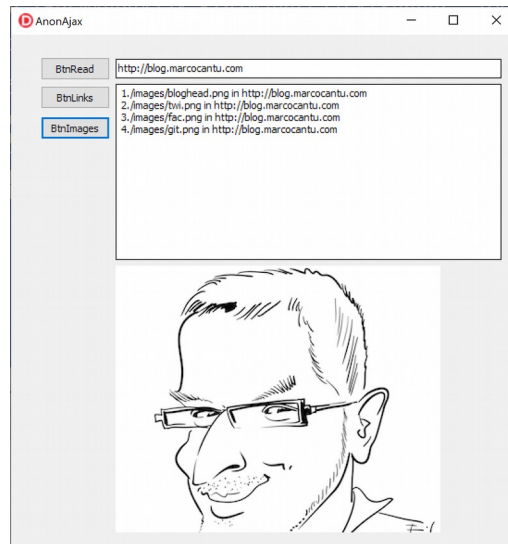
    Edurl.Text + AURL, AURL); // expand URL
    AjaxCall (Realurl,
    procedure (AResponseContent: TStringStream)
    begin
        Image1.Picture.Graphic.
            LoadFromStream (AResponseContent);
    end);
end;
end);
end);
end;

```

note This code snippet was the topic of a blog post of mine, “Anonymous, Anonymous, Anonymous” of September 2008, which attracted some comments, as you can see on: http://blog.marcocantu.com/blog/anonymous_3.html.

Beside the fact that the graphic only works in the case where you are loading a file with the same format as the one already in the Image component, the code and its result are both impressive. Notice in particular the numbering sequence, based on the capture of the `NHit` local variable. What happens if you press the button twice, in a fast sequence? Each of the anonymous methods will get a different copy of the `NHit` counter, and they might potentially be displayed out of sequence in the list, with the second thread starting to produce its output before the first. The result of using this last button can be seen in Figure 15.1.

Figure 15.1: The output of the triple nested anonymous method call to retrieve an image from a web page.



464 - 15: Anonymous Methods

16: reflection and attributes

Traditionally, compilers of strongly, statically typed languages, such as Pascal, provided little or no information about the available types at run-time. All the information about data types was visible only during the compilation phase.

The first version of Object Pascal broke with this tradition, by providing run time information for properties and other class members marked with a specific compiler directive, `published`. This feature was enabled for classes compiled with a specific setting `{$M+}` and it is the foundation of the streaming mechanism behind DFM files of the VCL (and FMX files of the FireMonkey library) and the way you work with the form and other visual designers. When it was first made available in Delphi 1, this feature was a completely new idea, which later other development tools adopted and extended it in several ways.

First, there were extensions to the type system (available only in Object Pascal) to account for method discovery and dynamic invocation in COM. This is still supported in Object Pascal by dispatch ID, applying methods to variants, and other COM-related features. Eventually COM support in Object Pascal was extended with

its own flavor of run time type information, but this is a topic well beyond the scope of a language book.

The advent of managed environments, such as Java and .NET, brought forward a very extensive form of run time type information, with detailed RTTI bound by the compiler to the executable modules and available for discovery by programs using those modules. This has the drawback of unveiling some of the program internals and of increasing the size of the modules, but it brings along new programming models that combine some of the flexibility of dynamic languages with the solid structure and the speed of strongly types ones.

Whether you like it or not (and this is indeed was the subject of intense debate at the time this feature was introduced) Object Pascal is slowly moving in the same direction, and the adoption of an extensive form of RTTI marks a very significant step in that direction. As we'll see, you can opt out of the RTTI, but if you don't you can leverage some extra power in your applications.

The topic is far from simple, so I will proceed in steps. We'll first focus on the new extended RTTI that's built into the compiler and the new classes of the `Rtti` unit that you can use to explore it. Second, I'll look at the new `TValue` structure and dynamic invocation. Third, I'll introduce custom attributes, a feature that parallels its .NET counterpart and let's you extend the RTTI information generated by the compiler. Only in the last part of the chapter will I try to get back to the reasons behind the extended RTTI and look at practical examples of its use.

Extended RTTI

The Object Pascal compiler generates by default a lot of extended RTTI information. This run time information includes all types, including classes and all other user defined types as well as the core data types predefined by the compiler and covers published fields as well as public ones, even protected and private elements. This is needed to be able to delve into the internal structure of any object.

A First Example

Before we look into the information generated by the compiler and the various techniques for accessing them, let me jump towards the conclusion and show you what can be done using RTTI. The specific example is very minimal and could have been

written with the older RTTI, but it should give you an idea of what I'm talking about (also considering that not all Object Pascal developers used the traditional RTTI explicitly).

Suppose you have a form with a button, like in the `RttiIntro` application project. You can write the following code to read the value of the control's `Text` property:

```
uses
    Rtti;

procedure TFormRttiIntro.BtnInfoClick(Sender: TObject);
var
    Context: TRttiContext;
begin
    Show (Context.
        GetType(TButton).
        GetProperty('Text').
        GetValue(Sender).ToString);
end;
```

The code uses the `TRttiContext` record to refer to information about the `TButton` type, from this type information to the RTTI data about a property, and this property data is used to refer to the actual value of the property, which is converted into a string. If you are wondering how this works, keep reading. My point here is that this approach can now be used not only to access a property dynamically, but also to read the values of fields, including private fields.

We can also change the value of a property, as the second button of the `RttiIntro` application project shows:

```
procedure TFormRttiIntro.BtnChangeClick(Sender: TObject);
var
    Context: TRttiContext;
    AProp: TRttiProperty;
begin
    AProp := Context.GetType(TButton).GetProperty('Text');
    AProp.SetValue(BtnChange,
        StringOfChar('*', Random(10) + 1));
end;
```

This code replaces the `Text` with a random number of `*`s. The difference from the code above is that it has a temporary local variable referring to the RTTI information for the property. Now that you have an idea what we are into, let's start from the beginning by checking the extended RTTI information generated by the compiler.

Compiler Generated Information

There is nothing you have to do to let the compiler add this extra information to your executable program (whatever its kind: application, library, package...). Just open a project and compile it. By default, the compiler generates Extended RTTI for all fields (including private ones) and for public and published methods and properties. You might be surprised by the fact that you get RTTI information for private fields, but this is required for dynamic operations like binary object serialization and tracing objects on the heap.

You can control the Extended RTTI generation according to a matrix of settings: On one axis you have the visibility and on the other the kind of member. The following table depicts the system default:

	Field	Method	Property
Private	x		
Protected	x		
Public	x	x	x
Published	x	x	x

Technically, the four visibility settings are indicated by using the following set type, declared in the `System` unit:

```
type
  TVisibilityClasses = set of (vcPrivate,
    vcProtected, vcPublic, vcPublished);
```

There are some ready to use constant values for this set indicating the default RTTI visibility settings applied to `TObject` and inherited by all other classes by default:

```
const
  DefaultMethodRttiVisibility = [vcPublic, vcPublished];
  DefaultFieldRttiVisibility = [vcPrivate..vcPublished];
  DefaultPropertyRttiVisibility = [vcPublic, vcPublished];
```

The information produced by the compiler is controlled by a new directive, `$RTTI`, which has a status indicating if the setting is for the given type or also for its descendants (`EXPLICIT` or `INHERITED`) followed by three specifiers to set the visibility for methods, fields, and properties. The default applied in the `System` unit is:

```
{$RTTI INHERIT
  METHODS(DefaultMethodRttiVisibility)
  FIELDS(DefaultFieldRttiVisibility)
  PROPERTIES(DefaultPropertyRttiVisibility)}
```

To completely disable the generation of extended RTTI for all of the members of your classes you can use the following directive:

```
| {$RTTI EXPLICIT METHODS([]) FIELDS([]) PROPERTIES([])}
```

note You cannot place the RTTI directive before the unit declaration, as it happens for other compiler directives, because it depends on settings defined in the `System` unit. If you do so, you'll receive an internal error message, which is not particularly intuitive. In any case, just move it after the unit statement.

When using this setting, consider it will be applied only to your code and a complete removal is not possible, as the RTTI information for the RTL and other library classes is already compiled into the corresponding DCUs and packages. Keep also in mind that the `$RTTI` directive doesn't cause any change on the traditional RTTI generated for published types: This is still produced regardless of the `$RTTI` directive.

note The RTTI processing classes, available in the `System.Rtti` unit and covered in the coming section, hook to the traditional RTTI and its `PTypeInfo` structure.

What you can do with this directive is stop the Extended RTTI being generated for your own classes. At the opposite end of the scale, you can also increase the amount of RTTI being generated, including private and protected methods and properties, if you wish (although it doesn't make a lot of sense).

The obvious effect of adding Extended RTTI information to an executable file is that the file will grow larger (which has the main drawback of a larger file to distribute, as the extra loading time and memory footprint is not so relevant). Now you can remove the RTTI from the units of your program, and this might have a positive effect... if you decide you don't want to use RTTI in your code. RTTI is a powerful technique, as you'll see in this chapter, and in most cases it is worth the extra executable size.

Weak and Strong Types Linking

What else could you do to reduce the size of the program? There is actually something you can do, even if its effect won't be big, it will be noticeable.

When evaluating the RTTI information available in the executable file, consider that what the compiler adds, the linker might remove. By default, classes and method not compiled in the program will not get the Extended RTTI (which would be quite useless), as they don't get the basic RTTI either. At the opposite end of the scale, if you want all Extended RTTI to be included and working, you need to link in even classes and methods you don't explicitly refer to in your code.

There are two compiler directives you can use to control the information being linked into the executable. The first, which is fully documented, is the `$weak-`

470 - 16: Reflection and Attributes

`LinkRTTI` directive. By turning it on, for types not used in the program, both the type itself and its RTTI information will be removed from the final executable.

Alternatively, you can force the inclusion of all types and their Extended RTTI using the `$StrongLinkTypes` directive. The effect on many programs is dramatic, with almost a two fold increase in the program size.

The RTTI Unit

If the generation of extended RTTI for all types is the first pillar for Reflection in Object Pascal, the second pillar is the ability to navigate this information in an easy and high level way, thanks to the `System.Rtti` unit. The third pillar, as we'll see later, is the support for custom attributes. But let me proceed one step at a time.

Traditionally, Object Pascal applications could (and still can) use the functions of the `System.TypeInfo` unit to access the “published” run time type information. This unit defines several low-level data structures and functions (all based on pointers and records) with a couple of higher level routines to make things a little easier.

The `Rtti` unit, instead, makes it very easy to work with the extended RTTI, providing a set of classes with proper methods and properties. To access the various objects, the entry point is the `TRttiContext` record structure, which has four methods to look for available types:

```
function GetType (ATypeInfo: Pointer): TRttiType; overload;  
function GetType (AClass: TClass): TRttiType; overload;  
function GetTypes: TArray<TRttiType>;  
function FindType (const AQualifiedName: string): TRttiType;
```

As you can see you can pass a class, a `PTTypeInfo` pointer obtained from a type, a qualified name (the name of the type decorated with the unit name, as in “*System.TObject*”), or retrieve the entire list of types, defined as an array of RTTI types, or more precisely as `TArray<TRttiType>`.

This last call is what I've used in the following listing, a simplified version of the code in the `TypesList` application project:

```
procedure TFormTypesList.BtnTypesListClick(Sender: TObject);  
var  
    AContext: TRttiContext;  
    TheTypes: TArray<TRttiType>;  
    AType: TRttiType;  
begin  
    TheTypes := AContext.GetTypes;  
    for AType in TheTypes do
```

```

    if AType.IsInstance then
        Show(AType.QualifiedName);
    end;

```

The `GetTypes` method returns the complete list of data types, but the program filters only the types representing classes. There are about a dozen other classes representing types in the unit.

note The `Rtti` unit refers to class types as “instances” or and “instance types” (as in `TRttiInstanceType`). This is a bit confusing, as we generally use the terms *instance* to refer to an actual object.

The individual objects in the types list are of classes which inherit from the `TRttiType` base class. Specifically, we can look for the `TRttiInstanceType` class type, rewriting the code above as in the following modified snippet:

```

    for AType in TheTypes do
        if AType is TRttiInstanceType then
            Show(AType.QualifiedName);

```

The actual code of the demo is a little more complex, as it populates a string list first, sorts the elements, and then populates a `ListView` control, using `BeginUpdate` and `EndUpdate` for optimization (and a `try` finally block around those to make sure the end operation is always performed):

```

var
    AContext: TRttiContext;
    TheTypes: TArray<TRttiType>;
    SList: TStringList;
    AType: TRttiType;
    STypeName: string;
begin
    ListView1.ClearItems;
    SList := TStringList.Create;
    try
        TheTypes := AContext.GetTypes;
        for AType in TheTypes do
            if AType.IsInstance then
                SList.Add(AType.QualifiedName);
        SList.Sort;
        ListView1.BeginUpdate;
        try
            for STypeName in SList do
                (ListView1.Items.Add).Text := STypeName;
            finally
                ListView1.EndUpdate;
            end;
        finally
            SList.Free;
        end;
    end;
end;

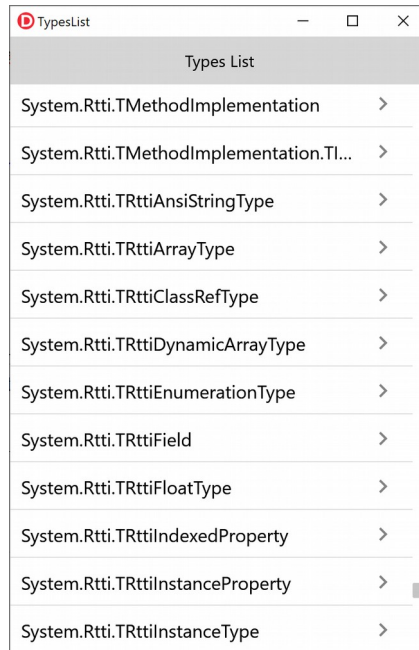
```


472 - 16: Reflection and Attributes

This code produces a rather long list with hundreds of data types, with the actual number depending on the platform and the version of the compiler, as you can see in Figure 16.1. Notice that the image lists types from the RTTI unit, covered in the next section.

Figure 16.1:

The output of the
TypesList application
project



The RTTI Classes in the Rtti Unit

In the following list, you can see the entire inheritance graph for the classes that derive from the abstract `TRttiObject` class and are defined in the `System.Rtti` unit:

```
TRttiObject (abstract)
  TRttiNamedObject
    TRttiType
      TRttiStructuredType (abstract)
        TRttiRecordType
        TRttiInstanceType
        TRttiInterfaceType
      TRttiOrdinalType
      TRttiEnumerationType
      TRttiInt64Type
      TRttiInvokableType
```

```

    TRttiMethodType
    TRttiProcedureType
    TRttiClassRefType
    TRttiEnumerationType
    TRttiSetType
    TRttiStringType
    TRttiAnsiStringType
    TRttiFloatType
    TRttiArrayType
    TRttiDynamicArrayType
    TRttiPointerType
    TRttiMember
    TRttiField
    TRttiProperty
    TRttiInstanceProperty
    TRttiIndexedProperty
    TRttiMethod
    TRttiParameter
    TRttiPackage
    TRttiManagedField

```

Each of these classes provides specific information about the given type. As an example, only a `TRttiInterfaceType` offers a way to access the interface GUID.

note In the first implementation of the `Rtti` unit there was no `RTTI` object to access indexed properties (like the `Strings[]` of a `TStringList`). This was later added and it is now available, making the run-time type information really complete.

RTTI Objects Lifetime Management and the `TRttiContext` record

If you look at the source code of the `BtnTypesListClick` method listed earlier, there is something that looks quite wrong. The `GetTypes` call returns an array of types, but the code doesn't free these internal objects.

The reason is that the `TRttiContext` record structure becomes the effective owner for all of the `RTTI` objects that are being created. When the record is disposed (that is, when it goes out of scope), an internal interface is cleared invoking its own destructor that frees all of the `RTTI` objects that were created through it.

The `TRttiContext` record actually has a dual role. On one side it controls the lifetime of the `RTTI` objects (as I just explained), on the other hand it caches `RTTI` information that is quite expensive to recreate with a search. That's why you might want to keep reference to the `TRttiContext` record alive for an extended period, allowing you to keep accessing the `RTTI` objects it owns without having to recreate them (again, the time consuming operation).

474 - 16: Reflection and Attributes

Internally the `TRttiContext` record uses a global pool of type `TRttiPool`, which uses a critical section to make its access thread safe.

note There are exceptions to the thread-safety of the RTTI pooling mechanism, described in some detail in the comments available in the `Rtti` unit itself.

So, to be more precise, the RTTI pool is shared among `TRttiContext` records, so the pooled RTTI objects are kept around while at least one `TRttiContext` record is in memory. To quote the comment in the unit:

```
{... working with RTTI objects without at least one context being alive  
is an error. Keeping at least one context alive should keep the Pool  
variable valid}
```

In other words, you have to avoid caching and keeping RTTI objects around after you've released the RTTI context. This is an example that leads to a memory access violation (again part of the `TypesList` application project):

```
function GetThisType (AClass: TClass): TRttiType;  
var  
    AContext: TRttiContext;  
begin  
    Result := AContext.GetType(AClass);  
end;  
  
procedure TFormTypesList.Button1Click(Sender: TObject);  
var  
    AType: TRttiType;  
begin  
    AType := GetThisType (TForm);  
    Show (AType.QualifiedName);  
end;
```

To summarize, the RTTI objects are managed by the context and you should not keep them around. The context in turn is a record, so it is disposed of automatically. You might see code that uses the `TRttiContext` in the following way:

```
AContext := TRttiContext.Create;  
try  
    // use the context  
finally  
    AContext.Free;  
end;
```

The pseudo-constructor and pseudo-destructor set the internal interface, that manages the actual data structures used behind the scenes, to `nil` cleaning up the pooling mechanism. However, as this operation is automatic for a local type such as a record, this is not needed, unless somewhere you refer to the context record using a pointer.

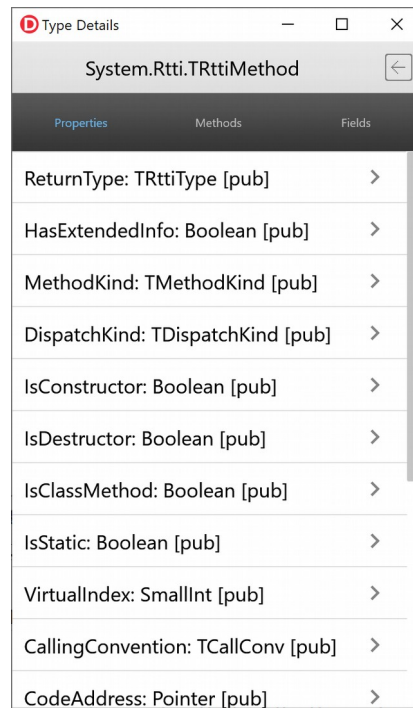
Displaying Class Information

The most relevant types you might want to inspect at run time are certainly the so-called structured types, that is instances, interfaces, and records. Focusing on instances, we can refer to the relationship among classes, by following the `BaseType` information available for instance types.

Accessing types is certainly an interesting starting point, but what is relevant and specifically new is the ability to learn about further details of these types, including their members. As you click on one of the types (here the `TRPopup` component class) the program displays a list of properties, methods, and fields of the type, in three pages of a tab control, as you can see in Figure 16.2.

Figure 16.2:

The detailed type information displayed by the `TypesList` application project for the class `TRttiMethod`



The unit of this secondary form, which can probably be adapted and expanded to be used as a generic type browser in other applications, has a method called `ShowTypeInfo` that walks through each property, method, and field of the given type, adding them to three separate list boxes with the indication of their visibility (*pub* for

476 - 16: Reflection and Attributes

private, *pro* for protected, *pub* for public, and *pbl* for published, as returned by a simple case statement in the `VisibilityToken` function):

```
procedure TFormTypeInfo.ShowTypeDetails(TypeName: string);  
var  
    AContext: TRttiContext;  
    AType: TRttiType;  
    AProperty: TRttiProperty;  
    AMethod: TRttiMethod;  
    AField: TRttiField;  
begin  
    AType := AContext.FindType(TypeName);  
    if not Assigned(AType) then  
        Exit;  
  
    LabelType.Text := AType.QualifiedName;  
    for AProperty in AType.GetProperties do  
        FormTypeInfo.LVProperties.Items.Add.Text := AProperty.Name +  
            ': ' + AProperty.PropertyType.Name + ' ' +  
            VisibilityToken (AProperty.Visibility);  
    for AMethod in AType.GetMethods do  
        LVMethods.Items.Add.Text := AMethod.Name + ' ' +  
            VisibilityToken (AMethod.Visibility);  
    for AField in AType.GetFields do  
        LVFields.Items.Add.Text := AField.Name + ': ' +  
            AField.FieldType.Name + ' ' +  
            VisibilityToken (AField.Visibility);  
end;
```

You could go ahead and extract further information from the types of these properties, get parameter lists of the methods and check the return type, and more. Here I don't want to build a complete RTTI browser but only give you a feeling of what can be achieved.

RTTI for Packages

Beside the methods you can use to access a type or the list of types, the record `TRttiContext` has another very interesting method, `GetPackages`, which returns a list of the run-time packages used by the current application. If you execute this method in an application compiled with no run time packages, all you get is the executable file itself. But if you execute it in an application compiled with run time packages, you'll get a list of those packages. From that point, you can delve into the types made available by each of the packages. Notice that in this case the list of types is much longer, as RTL and visual library types not used by the application are not removed by the smart linker.

If you use run time packages, you can also retrieve the list of types for each of the packages (and the executable file itself), by using code like:

```
var
  AContext: TRttiContext;
  APackage: TRttiPackage;
  AType: TRttiType;
begin
  for APackage in AContext.GetPackages do
    begin
      ListBox1.Items.Add( 'PACKAGE ' + APackage.Name);
      for AType in APackage.GetTypes do
        if AType.IsInstance then
          begin
            ListBox1.Items.Add( '    - ' + AType.QualifiedName);
          end;
        end;
      end;
    end;
  end;
```

note Packages in Object Pascal can be used to add components to the development environment, as we've seen in Chapter 11. However, packages can also be used at run-time, deploying a main executable with a few run-time packages, rather than a single, larger executable file. If you are familiar with Windows development, packages have a role similar to DLLs (and they technically are DLLs), or even more precisely as .NET assemblies. While packages play a very important role on Windows, they are not currently supported on mobile platforms (also due to operating systems application deployment limitations, like in iOS).

The TValue Structure

The new extended RTTI not only lets you browse the internal structure of a program but it also provides specific information, including property and field values. While the `TypeInfo` unit provided the `GetPropValue` function to access a generic property and retrieve a variant type with its value, the new `Rtti` unit uses a different structure for holding an untyped element, the `TValue` record.

This record can store almost any possible Object Pascal data type and does so by keeping track of the original data representation, by holding both the data and its data type. What it can do is read and write data in the given format. If you write an Integer to `TValue`, you can only read an Integer from it. If you write a string, you can read back the string.

What it cannot do is convert from one format to another. So even if a `TValue` has an `AsString` and an `AsInteger` method, you can use the former only if the data is representing is indeed a string, the second only if you originally assigned an integer to

478 - 16: Reflection and Attributes

it. For example, in this case you can use the `AsInteger` method and if you call the `IsOrdinal` method it will return `True`:

```
var
  v1: TValue;
begin
  v1 := 100;
  if v1.IsOrdinal then
    Log (IntToStr (v1.AsInteger));
```

However, you cannot use the `AsString` method, which would raise an *invalid type-cast* exception:

```
var
  v1: TValue;
begin
  v1 := 100;
  Log (v1.AsString);
```

If you need a string representation, though, you can use the `ToString` method, which has a large case statement trying to accommodate most data types:

```
var
  v1: TValue;
begin
  v1 := 100;
  Log (v1.ToString);
```

You can probably get a better understanding, by reading the words of Barry Kelly, a past Embarcadero R&D team member who worked on RTTI:

TValue is the type used to marshal values to and from RTTI-based calls to methods, and reads and writes of fields and properties. It's somewhat like Variant but far more tuned to the Object Pascal type system; for example, instances can be stored directly, as well as sets, class references, etc. It's also more strictly typed, and doesn't do (for example) silent string to number conversions.

Now that you better understand its role, let's look at the actual capabilities of the `TValue` record. It has a set of higher level methods for assigning and extracting the actual values, plus a set of low-level pointer based ones. I'll concentrate on the first group. For assigning values, `TValue` defines several `Implicit` operators, allowing you to perform a direct assignment as in the code snippets above:

```
class operator Implicit(const Value: string): TValue;
class operator Implicit(Value: Integer): TValue;
class operator Implicit(Value: Extended): TValue;
class operator Implicit(Value: Int64): TValue;
class operator Implicit(Value: TObject): TValue;
class operator Implicit(Value: TClass): TValue;
class operator Implicit(Value: Boolean): TValue;
```

What all these operators do is call the `From` generic class method:

```
class function From<T>(const Value: T): TValue; static;
```

When you call these class functions you need to specify the data type and also pass a value of that type, like the following code replacing the assignment of the value 100 of the previous code snippets:

```
v1 := TValue.From<Integer>(100);
```

This is a sort of universal technique for moving any data type into a `TValue`. Once the data has been assigned, you can use several methods to test its type:

```
property Kind: TTypeKind read GetTypeKind;
function IsObject: Boolean;
function IsClass: Boolean;
function IsOrdinal: Boolean;
function IsType<T>: Boolean; overload;
function IsArray: Boolean;
```

Notice that the generic `IsType` can be used for almost any data type.

There are corresponding method for extracting the data, but again you can use only the method compatible with the actual data stored in the `TValue`, as no conversion is taking place:

```
function AsObject: TObject;
function AsClass: TClass;
function AsOrdinal: Int64;
function AsType<T>: T;
function AsInteger: Integer;
function AsBoolean: Boolean;
function AsExtended: Extended;
function AsInt64: Int64;
function AsInterface: IInterface;
function AsString: string;
function AsVariant: Variant;
function AsCurrency: Currency;
```

Some of these methods double with a *Try* version that returns `False`, rather than raising an exception, in case of an incompatible data type. There are also some limited conversion methods, the most relevant of which are the generic `Cast` and the `ToString` function I've already used in the code:

```
function Cast<T>: TValue; overload;
function ToString: string;
```

Reading a Property with TValue

The importance of `TValue` lies in the fact that this is the structure used when accessing properties and field values using the extended RTTI and the `Rtti` unit. As an actual example of the use of `TValue`, we can use this record type to access both a

480 - 16: Reflection and Attributes

published property and a private field of a `TButton` object, as in the following code (part of the `RttiAccess` application project):

```
var
  Context: TRttiContext;
  AType: TRttiType;
  AProperty: TRttiProperty;
  AValue: TValue;
  AField: TRttiField;
begin
  AType := Context.GetType(TButton);
  AProperty := AType.GetProperty('Text');
  AValue := AProperty.GetValue(Sender);
  Show (AValue.AsString);

  AField := AType.GetField('FDesignInfo');
  AValue := AField.GetValue(Sender);
  Show (AValue.AsInteger.ToString);
end;
```

Invoking Methods

Not only does the new extended RTTI let you access values and fields, but it also provides a simplified way for calling methods. In this case you have to define a `TValue` element for each parameter of the method. There is a global `Invoke` function which you can call for executing a method:

```
function Invoke(CodeAddress: Pointer; const Args: Tarray<TValue>;
  CallingConvention: TCallConv; AResultType: PTypeInfo): TValue;
```

As a better alternative, there is a simplified `Invoke` overloaded method in the `TRttiMethod` class:

```
function Invoke(Instance: TObject;
  const Args: array of TValue): TValue; overload;
```

Two examples of invoking methods using this second simplified form (one returning a value and the second requiring a parameter) are part of the `RttiAccess` application project and listed below:

```
var
  Context: TRttiContext;
  AType: TRttiType;
  AMethod: TRttiMethod;
  Thevalues: array of TValue;
  AValue: TValue;
begin
  AType := context.GetType(TButton);
  AMethod := AType.GetMethod('ToString');
  Thevalues := [];
  AValue := AMethod.Invoke(Sender, Thevalues);
```

```

    Show(AValue.AsString);

    AType := Context.GetType(TForm1);
    AMethod := AType.GetMethod('Show');
    SetLength(TheValues, 1);
    TheValues[0] := AValue;
    AMethod.Invoke(self, TheValues);
end;

```

Using Attributes

The first part of this chapter gave you a good grasp of the extended RTTI generated by the Object Pascal compiler and of the RTTI access capabilities introduced by the new `Rtti` unit. In the second part of the chapter we can finally focus on one of the key reasons this entire architecture was introduced: the possibility to define custom attributes and extend the compiler-generated RTTI in specific ways. We'll look at this technology from a rather abstract perspective, and later focus on the reasons this is an important step forward for Object Pascal, by looking at practical examples.

What is an Attribute?

An attribute (in Object Pascal or C# terms) or an annotation (in Java jargon) is a comment or indication that you can add to your source code, applying it to a type, a field, a method, or a property) and the compiler will embed in the program. This is generally indicated with square brackets as in:

```

type
  [MyAttribute]
  TMyClass = class
    ...

```

By reading this information at design time in a development tool or at run time in the final application, a program can change its behavior depending on the values it finds.

Generally attributes are not used to change the actual core capabilities of a class of objects, but rather to let these classes specify further mechanisms they can participate in. Declaring a class as *serializable* doesn't affect its code in any way, but lets the serialization code know that it can operate on that class and how (in case you provide further information along with the attribute, or further attributes marking the class fields or properties).

482 - 16: Reflection and Attributes

This is exactly how the original and limited RTTI was used inside Object Pascal. Properties marked as published could show up in the object inspector, be streamed to a DFM file, and be accessed at run time. Attributes open up this mechanism to become much more flexible and powerful. They are also much more complex to use and easy to misuse, as are any powerful language features. I mean is don't throw away all the good things you know about Object Oriented Programming to embrace this new model, but complement one with the other.

As an example, an employee class will still be represented in a hierarchy as a derived class from a person class; an employee object will still have an ID for his or her badge; but you can “mark” or “annotate” the employee class as class that can be mapped to a database table or displayed by a specific runtime form. So we have inheritance (is-a), ownership (has-a), and annotations (marked-as) as three separate mechanism you can use when designing an application.

After you've seen the compiler features supporting custom attributes in Object Pascal and looked at some practical examples, the abstract idea I just mentioned should become more understandable, or at least that's my hope!

Attribute Classes and Attribute Declarations

How do you define a new attribute class (or attribute category)? You have to inherit from the new `TCustomAttribute` class available in the `System` unit:

```
type
  SimpleAttribute = class(TCustomAttribute)
  end;
```

The class name you give to the attribute class will become the symbol to use in the source code, with the optional exclusion of the *Attribute* postfix. So if you name your class `SimpleAttribute` you'll be able to use in the code an attribute called `Simple` or `SimpleAttribute`. For this is the reason the classic initial `τ` for Object Pascal classes is generally not used in case of attributes.

Now that we have defined a custom attribute, we can apply it to most of the symbols of our program: types, methods, properties, fields, and parameters. The syntax used for applying attributes is the attribute name within square brackets:

```
type
  [Simple]
  TMyClass = class(TObject)
  public
    [Simple]
    procedure One;
```

In this case I've applied the `Simple` attribute to the class as a whole and to a method. Beside a name, an attribute can support one or more parameters. The parameters passed to an attribute must match those indicated in the constructor of the attribute class, if any.

```
type
  ValueAttribute = class(TCustomAttributes)
    private
      FValue: Integer;
    public
      constructor Create(N: Integer);
      property Value: Integer read FValue;
    end;
```

This is how you can apply this attribute with one parameter:

```
type
  [Value(22)]
  TMyClass = class(TObject)
    public
      [Value(0)]
      procedure Two;
```

The attribute values, passed to its constructor, must be constant expressions, as they are resolved at compile time. That's why you are limited to just a few data types: ordinal values, strings, sets, and class references. On the positive side, you can have multiple overloaded constructors with different parameters.

Notice you can apply multiple attributes to the same symbol, as I've done in the `RttiAttrib` application project, which summarizes the code snippets of this section:

```
type
  [Simple][Value(22)]
  TMyClass = class(TObject)
    public
      [Simple]
      procedure One;
      [Value(0)]
      procedure Two;
    end;
```

What if you try to use an attribute that is not defined (maybe because of a missing `uses` statement)? Unluckily you get a very misleading warning message:

```
[DCC warning] RttiAttribMainForm.pas(44): w1025
  Unsupported language feature: 'custom attribute'
```

The fact this is a warning implies the attribute will be ignored, so you have to watch out for those warnings or even better treat the “unsupported language feature” warning like an error (something you can do in the Hints and Warnings page of the Project Options dialog box):

```
[DCC Error] RttiAttribMainForm.pas(38):
```

484 - 16: Reflection and Attributes

■ E1025 Unsupported language feature: 'custom attribute'

Finally, compared to other implementations of the same concept, there is currently no way to limit the scope of attributes, like declaring that an attribute can be applied to a type but not to a method. What is available in the editor, instead, is full support for attributes in the rename refactoring. Not only you can change the name of the attribute class, but the system will pick up when the attribute is used both in its full name and without the final “*attribute*” portion.

note Attributes refactoring was first mentioned by Malcolm Groves on his blog at: <http://www.malcolmgroves.com/blog/?p=554>

Browsing Attributes

Now this code would seem totally useless if there wasn't a way to discover which attributes are defined, and possibly inject a different behavior to an object because of these attributes. Let me start focusing on the first part. The classes of the `Rtti` unit let you figure out which symbols have associated attributes. This is code, extracted from the `RttiAttrib` application project shows the list of the attributes for the current class:

```
procedure TMyClass.One;
var
  Context: TRttiContext;
  Attributes: TArray<TCustomAttribute>;
  Attrib: TCustomAttribute;
begin
  Attributes := Context.GetType(ClassType).GetAttributes;
  for Attrib in Attributes do
    Form39.Log(Attrib.ClassName);
```

Running this code will print out:

```
SimpleAttribute
ValueAttribute
```

You can extend it by adding the following code to the `for-in` loop code to extract the specific value of the given attributes type:

```
if Attrib is ValueAttribute then
  Form39.Show ('-' + IntToStr
    (ValueAttribute(Attrib).Value));
```

What about fetching the methods with a given attribute, or with any attribute? You cannot filter the methods up front, but have to go through each of them, check their attributes, and see if it is relevant for you. To help in this process, I've written a function that checks if a method supports a given attribute:

```

type
  TCustomAttributeClass = class of TCustomAttribute;

function HasAttribute (AMethod: TRttiMethod;
  AttribClass: TCustomAttributeClass): Boolean;
var
  Attributes: TArray<TCustomAttribute>;
  Attrib: TCustomAttribute;
begin
  Result := False;
  Attributes := AMethod.GetAttributes;
  for Attrib in Attributes do
    if Attrib.InheritsFrom (AttribClass) then
      Exit (True);
end;

```

The HasAttribute function is called by the RttiAttrib program while checking for a given attribute or any of them:

```

var
  Context: TRttiContext;
  AType: TRttiType;
  AMethod: TRttiMethod;
begin
  AType := Context.GetType(TMyClass);

  for AMethod in AType.GetMethods do
    if HasAttribute (AMethod, SimpleAttribute) then
      Show (AMethod.Name);

  for AMethod in AType.GetMethods do
    if HasAttribute (AMethod, TCustomAttribute) then
      Show (AMethod.Name);

```

The effect is to list methods marked with the given attributes, as described by further Log calls which I've omitted from the listing above:

```

Methods marked with [Simple] attribute
One

Methods marked with any attribute
One
Two

```

Rather than simply describing attributes, what you generally do is add some independent behavior determined by the attributes of a class, rather than its actual code. As an example, I can inject a specific behavior in the previous code: The goal could be calling all methods of a class marked with a given attribute, considering them as parameterless methods:

```

procedure TForm39.BtnInvokeIfZeroClick(Sender: TObject);
var
  Context: TRttiContext;
  AType: TRttiType;

```

486 - 16: Reflection and Attributes

```
AMethod: TRttiMethod;  
ATarget: TMyClass;  
ZeroParams: array of TValue;  
begin  
  ATarget := TMyClass.Create;  
  try  
    AType := Context.GetType(ATarget.ClassType);  
    for AMethod in AType.GetMethods do  
      if HasAttribute (AMethod, SimpleAttribute) then  
        AMethod.Invoke(ATarget, ZeroParams);  
    finally  
      ATarget.Free;  
    end;  
  end;  
end;
```

What this code snippet does is create an object, grab its type, check for a given attribute, and invoke each method that has the `Simple` attribute. Rather than inheriting from a base class, implementing an interface, or writing specific code to perform the request, all we have to do to get the new behavior is mark one of more methods with a given attribute. Not that this example makes the use of attributes extremely obvious: for some common patterns in using attributes and some actual case studies you can refer to the final part of this chapter.

Virtual Methods Interceptors

This section covers a very advanced feature of Object Pascal and you might want to skip reading it, if you are just starting to learn Delphi's language. It is meant for more expert readers.

There is another relevant feature that was added after the extended RTTI was introduced, and it is the ability to *intercept* the execution of virtual methods of an existing class, by creating a proxy class for an existing object. In other words, you can take an existing object and change its virtual methods (a specific one or all of them at once).

Why would you want to do this? In a standard Object Pascal application, you probably would not use this feature. If you need an object with a different behavior, just change it or create a subclass. Things are different for libraries, because libraries should be written in a very generic way, knowing little about the objects they'll be able to manipulate, and imposing as little burden as possible on the objects themselves. This is the kind of scenario Virtual Methods Interceptors were added to Object Pascal.

note A very detailed blog post by Barry Kelly on Virtual Method Interceptors (to which I owe a lot) is in available at <http://blog.barrkel.com/2010/09/virtual-method-interception.html>.

Before we focus on possible scenarios, let me discuss the technology in itself. Suppose you have an existing class with at least a virtual method, like the following:

```

type
  TPerson = class
  ...
  public
    property Name: string read FName write SetName;
    property BirthDate: TDate read FBirthDate write SetBirthDate;
    function Age: Integer; virtual;
    function ToString: string; override;
  end;

function TPerson.Age: Integer;
begin
  Result := YearsBetween (Date, FBirthDate);
end;

function TPerson.ToString: string;
begin
  Result := FName + ' is ' + IntToStr (Age) + ' years old';
end;

```

Now what you can do is to create a `TVirtualMethodInterceptor` object (a new class defined in the `RTTI` unit) tied to the class subclass the object, changing the object's static class to the dynamic one:

```

var
  Vmi: TVirtualMethodInterceptor;
begin
  Vmi := TVirtualMethodInterceptor.Create(TPerson);
  Vmi.Proxify(Person1);

```

Once you have the `vmi` object you can *install* special handlers for its events (`OnBefore`, `OnAfter`, and `OnException`) using anonymous methods. These will be triggered before any virtual method call, after any virtual method call, and in case of an exception in a virtual method. These are the signatures for the three anonymous method types:

```

TInterceptBeforeNotify = reference to procedure(
  Instance: TObject; Method: TRttiMethod;
  const Args: TArray<TValue>; out DoInvoke: Boolean;
  out Result: TValue);
TInterceptAfterNotify = reference to procedure(
  Instance: TObject; Method: TRttiMethod;
  const Args: TArray<TValue>; var Result: TValue);
TInterceptExceptionNotify = reference to procedure(
  Instance: TObject; Method: TRttiMethod;
  const Args: TArray<TValue>; out RaiseException: Boolean;

```


488 - 16: Reflection and Attributes

```
|     TheException: Exception; out Result: TValue);
```

In each event you get the object, the method reference, the parameters, and the result (which might be already set or not). In the `OnBefore` event you can set `DoInvoke` parameter to disable the standard execution. In the `OnExcept` event you get information about the exception.

In the `InterceptBaseClass` application project, which uses the `TPerson` class above, I've intercepted the class virtual methods with this logging code:

```
| procedure TFormIntercept.BtnInterceptClick(Sender: TObject);  
| begin  
|     Vmi := TVirtualMethodInterceptor.Create(TPerson);  
|     Vmi.OnBefore := procedure(Instance: TObject; Method: TRttiMethod;  
|         const Args: TArray<TValue>; out DoInvoke: Boolean;  
|         out Result: TValue)  
|         begin  
|             Show('Before calling ' + Method.Name);  
|         end;  
|     Vmi.OnAfter := procedure(Instance: TObject; Method: TRttiMethod;  
|         const Args: TArray<TValue>; var Result: TValue)  
|         begin  
|             Show('After calling ' + Method.Name);  
|         end;  
|     Vmi.Proxify(Person1);  
| end;
```

Notice that the `vmi` object needs to be kept around at least until the `Person1` object is in use, or you'll be using a dynamic class that's not available any more and you'll be calling anonymous methods that have already been released. In the demo, I've saved it as a form field, just like the object to which it refers.

The program uses the object by calling its methods and checking the base class name:

```
|     Show ('Age: ' + IntToStr (Person1.Age));  
|     Show ('Person: ' + Person1.ToString);  
|     Show ('Class: ' + Person1.ClassName);  
|     Show ('Base Class: ' + Person1.ClassParent.ClassName);
```

Before you install the interceptor, the output is:

```
| Age: 26  
| Person: Mark is 26 years old  
| Class: TPerson  
| Base Class: TObject
```

After you install the interceptor, the output becomes:

```
| Before calling Age  
| After calling Age  
| Age: 26  
| Before calling ToString  
| Before calling Age
```

```

After calling Age
After calling ToString
Person: Mark is 26 years old
Class: TPerson
Base Class: TPerson

```

Notice that the class has the same name of the base class, but it is in fact a different one, the dynamic class created by the Virtual Method Interceptor. Although there has no official way to restore the class of the target object to the original one, the class itself is available in the Virtual Method Interceptor object and also as base class of the object. Still, you can use *brute force* to assign to the class data of the object (its initial four bytes) the correct class reference:

```

PPointer(Person1)^ := Vmi.OriginalClass;

```

As a further example, I've modified the `onBefore` code so that in case you are calling `Age` it returns a given value and skips executing the actual method:

```

Vmi.OnBefore := procedure(Instance: TObject; Method: TRttiMethod;
  const Args: TArray<TValue>; out DoInvoke: Boolean;
  out Result: TValue)
begin
  Show ( 'Before calling ' + Method.Name);
  if Method.Name = 'Age' then
    begin
      Result := 33;
      DoInvoke := False;
    end;
  end;
end;

```

The output changes from the version above, as follows (notice that the `Age` calls and the relative `onAfter` events are skipped):

```

Before calling Age
Age: 33
Before calling ToString
Before calling Age
After calling ToString
Person: Mark is 33 years old
Class: TPerson
Base Class: TPerson

```

Now that we have seen the technical details behind Virtual Methods Interceptor, we can get back to figure out in which scenarios you'd want to use this feature. Again, there is basically no reason to use this in a standard application. The focus, instead, is mostly for those who develop advanced libraries and need to implement a custom behavior for testing or processing objects.

For example, this could help building a unit testing library, although it would be limited to virtual methods only. You'd also possibly use this along with custom attributes to implement a coding style similar to Aspect Oriented Programming.

RTTI Case Studies

Now that I've covered the foundations of RTTI and the use of attributes it is worth looking into some real world situations in which using these technique will prove useful. There are many scenarios in which a more flexible RTTI and the ability to customize it through attributes is relevant, but I have no room for a long list of situations. What I'll do instead is guide you in the step-by-step development of two simple but significant examples.

The first demo program will showcase the use of attributes to identify specific information within a class. In particular, we want to be able to inspect an object of a class that declares to be part of an architecture and have a description and a unique ID referring to the object itself. This might come handy in several situations, like describing objects stored in a collection (either a generic or traditional one).

The second demo will be an example of streaming, specifically streaming a class to an XML file. I'll start from the classic approach of using the published RTTI, move to the new extended RTTI, and finally show how you can use attributes to customize the code and make it more flexible.

Attributes for ID and Description

If you want to have a couple of methods shared among many objects, the most classic approach was to define a base class with virtual methods and inherit the various objects from the base class, overriding the virtual methods. This is nice, but poses a lot of restrictions in terms of the classes which can participate in the architecture, as you have a fixed base class.

A standard technique to overcome this situation is to use an interface rather than a common base class. Multiple classes implementing the interface (but with no common ancestor class) can provide an implementation of the interface methods, which act very similarly to virtual methods.

A totally different style (with both advantages and disadvantages) is the use of attributes to mark participating classes and given methods (or properties). This opens up more flexibility, doesn't involve interfaces, but is based on a comparatively slow and error-prone run-time information look up, rather than a compile-time resolution. This means I'm not advocating this coding style over interfaces as a better approach, only as one that might be worth evaluating and interesting to use in some circumstances.

The Description Attribute Class

For this demo, I've defined an attribute with a setting indicating the element is it being applied to. I could have used three different attributes, but prefer to avoid polluting the attributes name space. This is the attribute class definition:

```
type
  TDescriptionAttrKind = (dakClass, dakDescription, dakId);

  DescriptionAttribute = class (TCustomAttribute)
  private
    FDak: TDescriptionAttrKind;
  public
    constructor Create (ADak: TDescriptionAttrKind = dakClass);
    property Kind: TDescriptionAttrKind read FDak;
  end;
```

Notice the use of the constructor with a default value for its only parameter, to let you use the attribute with no parameters.

The Sample Classes

Next I wrote two sample classes that use the attribute. Each class is marked with the attribute and has two methods marked with the same attribute customized with the different *kinds*. The first (TPerson) has the description mapped to the GetName function and uses its TObject.GetHashCode method to provide a (temporary) ID, re-declaring the method to apply the attribute to it (the method code is simply a call to the inherited version):

```
type
  [Description]
  TPerson = class
  private
    FBirthDate: TDate;
    FName: string;
    FCountry: string;
    procedure SetBirthDate(const value: TDate);
    procedure SetCountry(const value: string);
    procedure SetName(const value: string);
  public
    [Description (dakDescription)]
    function GetName: string;
    [Description (dakID)]
    function GetStringCode: Integer;
  published
    property Name: string read GetName write SetName;
    property BirthDate: TDate
      read FBirthDate write SetBirthDate;
    property Country: string read FCountry write SetCountry;
  end;
```

492 - 16: Reflection and Attributes

The second class (TCompany) is even simpler as it has its own values for the ID and the description:

```
type
[Description]
TCompany = class
private
  FName: string;
  FCountry: string;
  FID: string;
  procedure SetName(const Value: string);
  procedure SetID(const Value: string);
public
  [Description (dakDescription)]
  function GetName: string;
  [Description (dakID)]
  function GetID: string;
published
  property Name: string read GetName write SetName;
  property Country: string read FCountry write FCountry;
  property ID: string read FID write SetID;
end;
```

Even if there are similarities among the two classes they are totally unrelated in terms of hierarchy, common interface, or anything like that. What they share is the use of the same attribute.

The Sample Project and Attributes Navigation

The shared use of the attribute is used to display information about objects added to a list, declared in the main form of the program as:

```
| FObjectsList: TObjectList<TObject>;
```

This list is created and initialized as the program starts:

```
procedure TFormDescrAttr.FormCreate(Sender: TObject);
var
  APerson: TPerson;
  ACompany: TCompany;
begin
  FObjectsList := TObjectList<TObject>.Create;

  // add a person
  APerson := TPerson.Create;
  APerson.Name := 'Wiley';
  APerson.Country := 'Desert';
  APerson.BirthDate := Date - 1000;
  FObjectsList.Add(APerson);

  // add a company
  ACompany := TCompany.Create;
  ACompany.Name := 'ACME Inc.';
```

```

ACompany.ID := IntToStr (GetTickCount);
ACompany.Country := 'worldwide';
FObjectsList.Add(ACompany);

// add an unrelated object
FObjectsList.Add(TStringList.Create);

```

To display information about the objects (namely the ID and the description, if available) the program uses attributes discovery via RTTI. First, it uses a helper function to determine if the class is marked with the specific attribute:

```

function TypeHasDescription (aType: TRttiType): Boolean;
var
  Attrib: TCustomAttribute;
begin
  for Attrib in AType.GetAttributes do
    begin
      if (Attrib is DescriptionAttribute) then
        Exit (True);
    end;
  Result := False;
end;

```

note In this case you need to check for the full class name, `DescriptionAttribute`, and not only “Description”, which is the symbol you can use when applying the attribute.

If this is the case, the program proceeds by getting each attribute of each method, with a nested loop, and checking if this is the attribute we are looking for:

```

if TypeHasDescription (AType) then
  begin
    for AMethod in AType.GetMethods do
      for Attrib in AMethod.GetAttributes do
        if Attrib is DescriptionAttribute then
          ...
  end

```

At the core of the loop, the methods marked with attributes are invoked to read the results in two temporary strings (later added to the user interface):

```

if Attrib is DescriptionAttribute then
  case DescriptionAttribute(Attrib).Kind of
    dakClass: ; // ignore
    dakDescription:
      strDescr := AMethod.Invoke(AnObject, []).ToString;
    dakId:
      strID := AMethod.Invoke(AnObject, []).ToString;
  end

```

What the program fails to do is to check if an attribute is duplicated (that is, if there are multiple methods marked with the same attribute, a situation in which you might want to raise an exception).

494 - 16: Reflection and Attributes

Summing up all of the snippets of the previous page, this is the complete code of the `updateList` method:

```
procedure TFormDescrAttr.UpdateList;
var
  AnObject: TObject;
  Context: TRttiContext;
  AType: TRttiType;
  Attr: TCustomAttribute;
  AMethod: TRttiMethod;
  StrDescr, StrID: string;
begin
  for AnObject in FObjectsList do
    begin
      aType := context.GetType(AnObject.ClassInfo);
      if TypeHasDescription (AType) then
        begin
          for AMethod in AType.GetMethods do
            for Attr in AMethod.GetAttributes do
              if Attr is DescriptionAttribute then
                case DescriptionAttribute(Attr).Kind of
                  dakClass: ; // ignore
                  dakDescription:
                    // should check if duplicate attribute
                    StrDescr := aMethod.Invoke(
                      AnObject, []).ToString;
                  dakID:
                    StrID := AMethod.Invoke(
                      AnObject, []).ToString;
                end;
              // done looking for attributes
              // should check if we found anything
              with ListView1.Items.Add do
                begin
                  Text := STypeName;
                  Detail := StrDescr;
                end;
            end;
          end;
          // else ignore the object, could raise an exception
        end;
    end;
```

If this program produces rather uninteresting output, the way it is done is relevant, as I've marked some classes and two methods of those classes with an attribute, and have been able to process these classes with an external algorithm.

In other words, the classes themselves need no specific base class, no interface implementation nor any internal code to be part of the architecture, but only need to *declare* they want to get involved by using attributes. The full responsibility for managing the classes is in some external code.

XML Streaming

One interesting and very useful case for using RTTI is creating an “*external*” image of an object, for saving its status to a file or sending it over the wire to another application. Traditionally, the Object Pascal approach to this problem has been streaming the published properties of an object, the same approach used when creating DFM files. Now the RTTI lets you save the actual data of the object, its fields, rather than the external interface. This is more powerful, although it can lead to extra complexity, for example in the management of the data of internal objects. Again, the demo acts as a simple showcase of the technique and doesn't delve into all of its implications.

This examples comes in three versions, compiled in a single project for simplicity. The first is the traditional Object Pascal approach based on published properties, the second uses the extended RTTI and fields, the third uses attributes to customize the data mapping.

The Trivial XML Writer Class

To help with the generation of the XML, I've based the `xmlPersist` application project on an extended version of a `TTrivialXmlWriter` class I originally wrote in my Delphi 2009 Handbook to demonstrate the use of the `TTextWriter` class. Here I don't want to cover it again. Suffice to say that the class can keep track of the XML nodes it opens, thanks to a stack of strings, and close the XML nodes in a LIFO (Last In, First Out) order.

note The source code of the `TTrivialXmlWriter` class of Delphi 2009 Handbook can be found at <http://github.com/MarcoDelphiBooks/Delphi2009Handbook/tree/master/07/ReaderWriter>

To the original class I've added some limited formatting code and three methods for saving an object, based on the three different approaches I'm going to explore in this section. This is the complete class declaration:

```
type
  TTrivialXmlWriter = class
  private
    FWriter: TTextWriter;
    FNodes: TStack<string>;
    FOwnsTextWriter: Boolean;
  public
    constructor Create (AWriter: TTextWriter); overload;
    constructor Create (AStream: TStream); overload;
    destructor Destroy; override;
    procedure WriteStartElement (const SName: string);
    procedure WriteEndElement (Indent: Boolean = False);
```


496 - 16: Reflection and Attributes

```
procedure WriteString (const SValue: string);
procedure WriteObjectPublished (AnObj: TObject);
procedure WriteObjectRtti (AnObj: TObject);
procedure WriteObjectAttrib (AnObj: TObject);
function Indentation: string;
end;
```

To get an idea of the code, this is the `WriteStartElement` method, which uses the `Indentation` function for leaving twice as many spaces as the current number of nodes on the internal stack:

```
procedure TTrivialXmlWriter.WriteStartElement(
  const SName: string);
begin
  FWriter.Write (Indentation + '<' + SName + '>');
  FNodes.Push (SName);
end;
```

You'll find the complete code of the class in the project source code.

Classic RTTI-Based Streaming

After this introduction covering the support class, let me start from the very beginning, that is saving an object in an XML-based format using the classic RTTI for published properties.

The code of the `WriteObjectPublished` method is quite complex and requires a bit of explanation. It is based on the `TypeInfo` unit and uses the low-level version of the old RTTI to be able to get the list of published properties for a given object (the `AnObj` parameter), with code like:

```
NProps := GetTypeData(AnObj.ClassInfo)^.PropCount;
GetMem(PropList, NProps * SizeOf(Pointer));
GetPropInfos(AnObj.ClassInfo, PropList);
for I := 0 to NProps - 1 do
  ...
```

What this does is ask for the number of properties, allocate a data structure of the proper size, and fill the data structure with information about the published properties. In case you are wondering could you write this low-level code? Well you've just found a very good reason why the new RTTI was introduced. For each property, the program extracts the value of properties of numeric and string types, while it extracts any sub-object and acts recursively on it:

```
StrPropName := UTF8ToString (PropList[i].Name);
case PropList[i].PropType^.Kind of
  tkInteger, tkEnumeration, tkString, tkUString, ...:
begin
  WriteStartElement (StrPropName);
  WriteString (GetPropValue(AnObj, StrPropName));
  WriteEndElement;
```

```

end;
tkClass:
begin
  InternalObject := GetObjectProp(AnObj, StrPropName);
  // recurse in subclass
  WriteStartElement (StrPropName);
  WriteObjectPublished (InternalObject as TPersistent);
  WriteEndElement (True);
end;
end;

```

There is some extra complexity, but for the sake of the example and to give you an idea of the traditional approach, that should be enough.

To demonstrate the effect of the program I've written two classes (TCompany and TPerson) adapted from the previous example. This time, however, the company can have a person assigned to an extra property, called Boss. In the real world this would be more complex, but for this example it is a reasonable assumption. These are the published properties of the two classes:

```

type
  TPerson = class (TPersistent)
  ..
  published
    property Name: string read FName write FName;
    property Country: string read FCountry write FCountry;
  end;

  TCompany = class (TPersistent)
  ..
  published
    property Name: string read FName write FName;
    property Country: string read FCountry write FCountry;
    property ID: string read FID write FID;
    property Boss: TPerson read FPerson write FPerson;
  end;

```

The main form of the program has a button used to create and connect two objects of these two classes and saving them to an XML stream, which is later displayed.

The streaming section has the following code:

```

SS := TStringStream.Create;
XmlWri := TTrivialXmlWriter.Create (SS);
XmlWri.WriteStartElement('company');
XmlWri.WriteObjectPublished(aCompany);
XmlWri.WriteEndElement;

```

The result is an XML file like:

```

<company>
  <Name>ACME Inc.</Name>
  <Country>worldwide</Country>
  <ID>29088851</ID>
  <Boss>

```

498 - 16: Reflection and Attributes

```
<Name>Wiley</Name>
<Country>Desert</Country>
</Boss>
</company>
```

Streaming Fields With Extended RTTI

With the high-level RTTI available in Object pascal, I could have converted this old program to use the extended RTTI for accessing the published properties. What I'm going to do, instead, is to use it for saving the internal representation of the object, that is, its private data fields. Not only am I doing something more hard-core, but I'm doing it with much higher-level code. The complete code of the `writeObjectRtti` method is the following:

```
procedure TTrivialXmlWriter.writeObjectRtti(AnObj: TObject);
var
  AContext: TRttiContext;
  AType: TRttiType;
  AField: TRttiField;
begin
  AType := AContext.GetType (AnObj.ClassType);
  for AField in AType.GetFields do
    begin
      if AField.FieldType.IsInstance then
        begin
          writeStartElement (AField.Name);
          writeObjectRtti (AField.GetValue(AnObj).AsObject);
          writeEndElement (True);
        end
      else
        begin
          writeStartElement (AField.Name);
          writeString (AField.GetValue(AnObj).ToString);
          writeEndElement;
        end;
      end;
    end;
  end;
```

The resulting XML is somewhat similar, but somehow less clean as field names are generally less readable than property names:

```
<company>
  <FName>ACME Inc.</FName>
  <FCountry>Worldwide</FCountry>
  <FID>29470148</FID>
  <FPerson>
    <FName>Wiley</FName>
    <FCountry>Desert</FCountry>
  </FPerson>
</company>
```

Another big difference, though, is that in this case the classes didn't need to inherit from the `TPersistent` class or be compiled with any special option.

Using Attributes to Customize Streaming

Beside the problem with the tag names, there is another issue I haven't mentioned. Using XML tag names which are actually compiled symbols is far from a good idea. Also, in the code there is no way to exclude some properties or fields from XML-base streaming.

note Object Pascal properties streaming can be controlled using the `stored` directive, which can be read using the `TypeInfo` unit. Still, this solution is far from simple and clean, even if the DFM streaming mechanism uses it effectively.

These are issues we can address using attributes, although the drawback will be having to use them quite heavily in the declaration of our classes, a coding style I don't like much. For the new version of the code, I've defined an attribute constructor with an optional parameter:

```
type
  XmlAttribute = class (TCustomAttribute)
  private
    FTag: string;
  public
    constructor Create (StrTag: string = '');
    property TagName: string read FTag;
  end;
```

The attributes-based streaming code is a variation of the last version based on the extended RTTI. The only difference is that now the program calls the `CheckXmlAttr` helper function to verify if the field has the `xml` attribute and the (optional) tag name decoration:

```
procedure TTrivialXmlWriter.WriteObjectAttrib(AnObj: TObject);
var
  AContext: TRttiContext;
  AType: TRttiType;
  AField: TRttiField;
  StrTagName: string;
begin
  AType := AContext.GetType (AnObj.ClassType);
  for AField in AType.GetFields do
    begin
      if CheckXmlAttr (AField, StrTagName) then
        begin
          if AField.FieldType.IsInstance then
            begin
              WriteStartElement (StrTagName);
              WriteObjectAttrib (AField.GetValue(AnObj).AsObject);
```

500 - 16: Reflection and Attributes

```
        WriteEndElement (True);
    end
    else
    begin
        WriteStartElement (StrTagName);
        WriteString (AField.GetValue(AnObj).ToString);
        WriteEndElement;
    end;
end;
end;
end;
```

The most relevant code is in the `CheckXmlAttribute` helper function:

```
function CheckXmlAttribute (AField: TRttiField;
    var StrTag: string): Boolean;
var
    Attrib: TCustomAttribute;
begin
    Result := False;
    for Attrib in AField.GetAttributes do
        if Attrib is XmlAttribute then
            begin
                StrTag := XmlAttribute(Attrib).TagName;
                if StrTag = '' then // default value
                    StrTag := AField.Name;
                Exit (True);
            end;
    end;
end;
```

Fields without the XML attribute are ignored and the tag used in the XML output is customizable. To demonstrate this, the program has the following classes (this time I've omitted the published properties from the listing, as they are not relevant):

```
type
    TAttrPerson = class
    private
        [xml ('Name')]
        FName: string;
        [xml]
        FCountry: string;
        ...

    TAttrCompany = class
    private
        [xml ('CompanyName')]
        FName: string;
        [xml ('Country')]
        FCountry: string;
        FID: string; // omitted
        [xml ('TheBoss')]
        FPerson: TAttrPerson;
        ...
```

With these declarations, the XML output will look like the following (notice the tag name, the fact the ID is omitted, and the (bad looking) default name for the FCountry field):

```
<company>
  <CompanyName>ACME Inc.</CompanyName>
  <Country>Worldwide</Country>
  <TheBoss>
    <Name>Wiley</Name>
    <FCountry>Desert</FCountry>
  </TheBoss>
</company>
```

The difference here is we can be very flexible about which fields to include and how to name them in the XML, something the previous versions didn't allow.

Even if this is just a very skeletal implementation, I think that giving you the opportunity to see the final version being created step by step starting with the classic RTTI has given you a good feeling of the differences among the various techniques. What is important to keep in mind, in fact, is that it is not a given that using attributes will be always the best solution! On the other hand, it should be clear that RTTI and attributes add a lot of power and flexibility in any scenario in which you need to inspect the structure of an unknown object at run time.

Other RTTI-Based Libraries

To conclude this chapter, I'd like to point out the fact that there are several libraries, both part of the product and from third parties, that have started leveraging the extended RTTI. One such example is the binding expressions mechanism that sits behind visual live bindings. You can create a binding expression, assign it an expression (that is a string of text with operations like concatenation or addition), and have the expression refer to an external object and its field.

Even if I don't want to delve too much into this topic, which is really a specific library and not part of the language or the core system, I think a short listing can give you an idea:

```
var
  BindExpr: TBindingExpression;
  Pers: TPerson;
begin
  Pers := TPerson.Create;
  Pers.Name := 'John';
  Pers.City := 'San Francisco';

  BindExpr := TBindingExpressionDefault.Create;
  BindExpr.Source := 'person.name + " lives in " + person.city');
```

502 - 16: Reflection and Attributes

```
BindExpr.Compile([
  TBindingAssociation.Create(Pers, 'person')]);
Show (BindExpr.Evaluate.GetValue.ToString);

Pers.Free;
BindExpr.Free;
end;
```

Notice that the advantage here comes from the fact you can change the expression at run-time (although in the specific snippet above it is a constant string). The expression can come from an edit, or can be picked dynamically from several possible expressions. It is first assigned to the `TBindingExpression` object and then analyzed and *compiled* (that is transformed into a symbolic form, not assembly code) at run-time with the `Compile` call. It will then use RTTI when executed access the `TPerson` object.

The drawback is this approach makes the expression evaluation significantly slower than the execution of similar compiled Object Pascal source code. In other words, you have to balance the reduced performance with the increased flexibility. Also the Visual Live Binding model built on top of this makes for a very nice and easy developer experience.

17: tobject and the system unit

At the heart of any Object Pascal language application there is a hierarchy of classes. Every class in the system is ultimately a subclass of the `TObject` class, so the whole hierarchy has a single root. This allows you to use the `TObject` data type as a replacement for the data type of any class type in the system.

The `TObject` class is defined in a core RTL unit called `system`, which has such an important role that it is automatically included in every compilation. While I won't cover all `system` unit classes and other `System` unit functions, there are a few worth delving into and `TObject` is certainly the most important one.

note It could be debated at length if a core system class like `TObject` is part of the language or if it is part of the Run Time Library (RTL). The same is true for other features of the `system` unit, a unit so critical that it is automatically included in the compilation of any other unit. (It is actually illegal to add it to a `uses` statement.) Such a debate would be rather futile, though, so I'll just leave it for another time.

The TObject Class

As I've just mentioned the `TObject` class is a very special one, as all other classes inherit from it. When you are declaring a new class, in fact, if you are not indicating a base class, the class will automatically inherit from `TObject`. In programming language terms this type of scenario is called a *singly-rooted class hierarchy*, a feature Object Pascal shared with C#, Java, and quite a few other modern programming languages. The notable exception is C++, which has no concept of a single base class and allows you to define multiple totally separate class hierarchies.

This base class is not a class you'd use directly by creating instances of it. However it is a class you'll easily end up using a lot. Every time you need a variable that can hold an object of any type, you declare it of the `TObject` type. A good example of this usage is in component libraries event handlers, that very often have `TObject` as the type of the first parameter, generally called `Sender`. This means *any* object of *any* actual class. Many generic collection are also collections of objects, and there are several scenarios in which the `TObject` type is used directly. In the following sections I'm going to touch on a few of the features of this class, that are available to all classes in the system.

Construction and Destruction

Although it makes little sense to create a `TObject` directly, the constructor and destructor of this class are important, as they are automatically inherited by all other classes. If you define a class without a constructor, you can still call `Create` on it, invoking the `TObject` constructor, which is an empty method (as there is nothing to initialize in this base class). This `Create` constructor is non-virtual, and you totally replace it in your classes, unless this do-nothing constructor is good enough. Calling the base class constructor is a good practice for any subclass, even if a direct call to `TObject.Create` is not particularly useful.

note I've underlined this is a non-virtual constructor because there is another core library class, `TComponent`, that defines a virtual constructor, instead. The `TComponent` class virtual constructor plays a key role in the streaming system, covered in the next chapter.

For destroying an object, the `TObject` class has a `Free` method (which eventually calls the `Destroy` destructor). I've covered these in detail in Chapter 13, along with many suggestions on correct memory usage, so there is no need to reiterate them here.

Knowing About an Object

An interesting group of methods of the `TObject` class is those returning information about the type. The most commonly used ones are the `ClassType` and `ClassName` methods. The `ClassName` method returns a string with the name of the class. Because it is a class method (like a large number of the `TObject` class methods), you can apply it both to an object and to a class. Suppose you have defined a `TButton` class and a `Button1` object of that class. Then the following statements have the same effect:

```
Text := Button1.ClassName;
Text := TButton.ClassName;
```

Of course, you can also apply those to a generic `TObject`, and you won't get `TObject` information, but information about the specific class of the object currently assigned to the variable. For example, in the `OnClick` event handler of a button, calling:

```
Text := Sender.ClassName;
```

would likely return the same of the lines above, that is the string `'TButton'`. This is because the class name is determined at run-time (by the specific object itself) and not by the compiler (which will only think this is a `TObject` object).

While getting the class name can be handy for debugging, logging, and showing class information in general, it is often more important to access the class reference of the class. As an example, it is better compare two class references than the strings with the class names. We can get class references with the `ClassType` method, while the `ClassParent` method returns a class reference to the base class of the current one, allowing to navigate to the base classes list. The only exception is that the method returns `nil` for `TObject` (as it has no parent class). Once you have a class reference, you can use it to call any class method, including the `ClassName` method.

Another very interesting method returning information about a class is `InstanceSize`, which returns the run-time size of an object, that is the amount of memory required by its fields (and those inherited from base classes). This is a feature used internally when the system needs to allocate a new instance of the class.

note Although you might think that the `SizeOf` global function also provides this information, that function actually returns the size of an object reference—a pointer, which is invariably four or eight bytes, depending on the target platform—instead of the size of the object itself.

More Methods of the TObject Class

There are other methods of the `TObject` class you can apply to any object (and also to any class or class reference, because they are class methods). Here is a partial list, with a short description:

- `ClassName` returns a string with the name of the class, for display.
- `ClassNameIs` checks the class name against a value.
- `ClassParent` returns a class reference to the parent class of the current class or object's class. You can navigate from `ClassParent` to `ClassParent`, until you reach the `TObject` class itself, in which this method returns `nil`.
- `ClassInfo` returns a pointer to the internal, low-level Run Time Type Information (RTTI) of the class. This was used in the early days of the `TypeInfo` unit, but it is now replaced by the capabilities of the `RTTI` unit, as covered in Chapter 16. Internally, this is still how the class RTTI is fetched.
- `ClassType` returns a reference to the object's class (this cannot be applied directly to a class, only to an object).
- `InheritsFrom` tests whether the class inherits (directly or indirectly) from a given base class (this is very similar to the `is` operator, and actually how the `is` operator is ultimately implemented.).
- `InstanceSize` returns the size of the object's data in bytes. This is a sum of the fields, plus some extra special reserved bytes (including for example the class reference). Notice, once more thing, this is the instance size, while the reference to an instance is only as long as a pointer (4 or 8 bytes, depending on the platform).
- `UnitName` returns the name of the unit in which the class is defined, which could be useful for describing a class. The class name, in fact, is not unique in the system. As we saw in the last chapter, only the qualified class name (made of the unit name and the class name, separated by a dot) is unique in an application.
- `QualifiedClassName` returns this combination of unit and class name, a value that is indeed unique in a running system.

These methods of `TObject` are available for objects of every class, since `TObject` is the common ancestor class of every class. Here is how we can use these methods to access class information:

```
procedure TSenderForm.ShowSender(Sender: TObject);
begin
    Memo1.Lines.Add ('Class Name: ' + Sender.ClassName);

    if Sender.ClassParent <> nil then
```

```

Memo1.Lines.Add ('Parent Class: ' + Sender.ClassParent.ClassName);
Memo1.Lines.Add ('Instance Size: ' + IntToStr (Sender.InstanceSize));

```

The code checks to see whether the `ClassParent` is `nil` in case you are actually using an instance of the `TObject` type, which has no base type. You can use other methods to perform tests. For example, you can check whether the `Sender` object is of a specific type with the following code:

```

if Sender.ClassType = TButton then ...

```

You can also check if the `Sender` parameter corresponds to a given object, with this test:

```

if Sender = Button1 then...

```

Instead of checking for a particular class or object, you'll generally need to test the type compatibility of an object with a given class; that is, you'll need to check whether the class of the object is a given class or one of its subclasses. This lets you know whether you can operate on the object with the methods defined for the class. This test can be accomplished using the `InheritsFrom` method, which is also called when you use the `is` operator. The following two tests are equivalent:

```

if Sender.InheritsFrom (TButton) then ...
if Sender is TButton then ...

```

Showing Class Information

Once you have a class reference, you can add to its description (or display information) a list of all of its base classes. In the following code snippets the base classes of `MyClass` are added to a `Listbox` control:

```

ListParent.Items.Clear;
while MyClass.ClassParent <> nil do
  begin
    MyClass := MyClass.ClassParent;
    ListParent.Items Add (MyClass.ClassName);
  end;

```

You'll notice that we use a class reference at the heart of the `while` loop, which tests for the absence of a parent class (in which case the current class is `TObject`). Alternatively, we could have written the `while` statement in either of the following ways:

```

while not MyClass.ClassNameIs ('TObject') do... // slow, error prone
while MyClass <> TObject do... // fast, and readable

```

TOBJECT'S Virtual Methods

While the structure of the `TObject` class has remained quite stable since the early days of the Object Pascal language, at one point it saw the addition of three extremely useful virtual methods. These are methods can be called on any object, like any other `TObject` method, but the relevance is that these are methods you are supposed to override and redefine in your own classes.

note If you've used the .NET framework you'll immediately recognize these methods are part of the `System.Object` class of the C# base class library. Similar methods are used for the base classes available in Java, are commonly used in JavaScript, and in other languages. The origin of some of them, like that of `toString`, can be traced back to Smalltalk, which is considered the first OOP language.

The ToString Method

The `ToString` virtual function is a placeholder for returning the textual representation (a description or even a serialization) of a given object. The default implementation of the method in the `TObject` class returns the class name:

```
function TObject.ToString: string;  
begin  
    Result := ClassName;  
end;
```

Of course, this is far from useful. In theory, each class should provide a way to describe itself to a user, for example when an object is added to a visual list. Some of the classes in the run time library override the `ToString` virtual function, like `TStringBuilder`, `TStringWriter`, and the `Exception` class, to return the messages in a list of exceptions (as covered in the section “The InnerException Mechanism” of Chapter 9).

Having a standard way to return the string representation of any object is quite an interesting idea, and I recommend you to take advantage of this core feature of the `TObject` class, treating it like a language feature.

note Notice that the `ToString` method “semantically overloads” the “parse token String” or `toString` symbol defined in the `Classes` unit. For this reason that symbol referenced as `Classes.toString`.

The Equals Method

The `Equals` virtual function is a placeholder for checking if two objects have the same logical value, a different operation than checking if two variables refer to the

same object in memory, something you can achieve with the = operator. However, and this is really confusing, the default implementation does exactly that, for the lack of a better way:

```
function TObject.Equals(Obj: TObject): Boolean;
begin
    Result := Obj = self;
end;
```

An example of the use of this method (with a proper override) is in the TStrings class, in which the Equals method compares the number of strings in the list and the content of the actual strings one by one.

A section of the library in which this technique is significantly used is generics support, in particular in the Generics.Default and Generics.Collections units. In general it is important for a library or framework to define the concept of object “value equivalence” separately from object identity. Having a standard mechanism for comparing objects “by value” is a big advantage.

The GetHashCode Method

The GetHashCode virtual function is another placeholder borrowed from the .NET framework to let each class calculate the hash code for its objects. The default code returns a seemingly random value, the address of the object itself:

```
function TObject.GetHashCode: Integer;
begin
    Result := Integer(self);
end;
```

note With the address of the objects being created generally taken from a limited set of heap areas, the distribution of these number is not even, and this can adversely affect a hashing algorithm. It is highly recommended to customize this method creating a hash based on logical values with a good hash distribution based on the data inside the object, rather than its address. Dictionaries and other data structure rely on hash values, improving the hash distribution can lead to significantly better performance.

The GetHashCode virtual function is used by some collection classes that support hash tables and as a way to optimize some code, like TDictionary <T>.

Using TObject Virtual Methods

Here is an example based on some of the TObject virtual methods. The example has a class that overrides two of these methods:

```
type
    TAnyObject = class
```

510 - 17: TObject and the System Unit

```
private
  FValue: Integer;
  FName: string;
public
  constructor Create (AName: string; AValue: Integer);
  function Equals(Obj: TObject): Boolean; override;
  function ToString: string; override;
end;
```

In the implementation of the three methods I simply had to change a call to `GetType` with that to `ClassType`:

```
constructor TAnyObject.Create(AName: string; AValue: Integer);
begin
  inherited Create;
  FName := AName;
  FValue := AValue;
end;

function TAnyObject.Equals(Obj: TObject): Boolean;
begin
  Result := (Obj.ClassType = self.ClassType) and
    ((Obj as TAnyObject).Value = self.Value);
end;

function TAnyObject.ToString: string;
begin
  Result := Name;
end;
```

Notice that objects are considered equal if they are of the same exact class and their value matches, while their string representation includes only the name field. The program creates some objects of this class as its starts:

```
procedure TFormSystemObject.FormCreate(Sender: TObject);
begin
  Ao1 := TAnyObject.Create ('Ao1', 10);
  Ao2 := TAnyObject.Create ('Ao2 or Ao3', 20);
  Ao3 := ao2;
  Ao4 := TAnyObject.Create ('Ao4', 20);
  ...
```

Notice that two references (Ao2 and Ao3) point to the same object in memory, and that the last object (Ao4) has the same numerical value. The program has a user interface that lets a user select any two the items and compare the selected objects, both using `Equals` and doing a direct reference comparison.

Here are some of the results:

```
Comparing Ao1 and Ao4
Equals: False
Reference = False

Comparing Ao2 and Ao3
```

```
Equals: True
Reference = True
```

```
Comparing Ao3 and Ao4
Equals: True
Reference = False
```

The program has another button used to test some of these methods for the button itself:

```
var
  Btn2: TButton;
begin
  Btn2 := BtnTest;
  Log ('Equals: ' +
    BoolToStr (BtnTest.Equals (Btn2), True));
  Log ('Reference = ' +
    BoolToStr (BtnTest = Btn2, True));
  Log ('GetHashCode: ' +
    IntToStr (BtnTest.GetHashCode));
  Log ('ToString: ' + BtnTest.ToString);
end;
```

The output is the following (with a hash value that changes upon execution):

```
Equals: True
Reference = True
GetHashCode: 28253904
ToString: TButton
```

TObject Class Summary

As a summary, this is the complete interface of the TObject class in the latest version of the compiler (with most of the IFDEF and low-level overloads omitted, along with private and protected sections):

```
type
  TObject = class
  public
    constructor Create;
    procedure Free;
    procedure DisposeOf;
    class function InitInstance(Instance: Pointer): TObject;
    procedure CleanupInstance;
    function ClassType: TClass; inline;
    class function ClassName: string;
    class function ClassNameIs(const Name: string): Boolean;
    class function ClassParent: TClass;
    class function ClassInfo: Pointer; inline;
    class function InstanceSize: Integer; inline;
    class function InheritsFrom(AClass: TClass): Boolean;
    class function MethodAddress(const Name: string): Pointer;
```



```

class function MethodName(Address: Pointer): string;
class function QualifiedClassName: string;
function FieldAddress(const Name: string): Pointer;
function GetInterface(const IID: TGUID; out Obj): Boolean;
class function GetInterfaceEntry(
  const IID: TGUID): PInterfaceEntry;
class function GetInterfaceTable: PInterfaceTable;
class function UnitName: string;
class function UnitScope: string;
function Equals(Obj: TObject): Boolean; virtual;
function GetHashCode: Integer; virtual;
function ToString: string; virtual;
function SafeCallException(ExceptObject: TObject;
  ExceptAddr: Pointer): HRESULT; virtual;
procedure AfterConstruction; virtual;
procedure BeforeDestruction; virtual;
procedure Dispatch(var Message); virtual;
procedure DefaultHandler(var Message); virtual;
class function NewInstance: TObject; virtual;
procedure FreeInstance; virtual;
destructor Destroy; virtual;
public
  property Disposed: Boolean read GetDisposed;
end;

```

Unicode and Class Names

Overloaded methods like `MethodAddress` and `FieldAddress` can take either a `UnicodeString` (UTF-16, as usual) or a `ShortString` parameter that is treated as a UTF-8 string. In fact, the versions taking a normal Unicode string, convert them by calling the function `UTF8EncodeToShortString`:

```

function TObject.FieldAddress(const Name: string): Pointer;
begin
  Result := FieldAddress(UTF8EncodeToShortString(Name));
end;

```

Since Unicode support was introduced in the language, the class names in Object Pascal internally use the `ShortString` representation (an array of one-byte characters), but with an UTF-8 encoding rather than the traditional ANSI encoding of the `ShortString` type. This happens both at the `TObject` level and at the RTTI level.

For example, the `ClassName` method is implemented with some really low level code as follows:

```

class function TObject.ClassName: string;
begin
  Result := UTF8ToString (
    PShortString (PPointer (
      Integer(Self) + vmtClassName)^)^);
end;

```

```
| end;
```

Similarly in the `TypeInfo` unit, all the functions accessing class names convert the internal UTF-8 `ShortString` representations to a `UnicodeString`. Something similar happens for property names.

The System Unit

While the `TObject` class has clearly a fundamental role for the language, making it very difficult to say if it is part of the language or of the run-time library, there are other low level classes in the `system` unit that constitute a fundamental and integrated part of the compiler support. Most of the content of this unit, though, is made of low level data structures, simple records structures, functions and procedures, and a few classes.

Here I'm going to focus mostly on the classes, but it is undeniable that many other features in the `system` unit are key to the language. For example, the `system` unit defines so-called “intrinsic” functions, that don't have actual code but are resolved directly by the compiler. An example is `sizeof`, which the compiler directly replaces with the actual size of the data structure that was passed as parameter.

You can gain an idea of the special role of the `system` unit by reading the comment added to its inception (mostly to explain why browsing system symbols leads to this unit... but not the symbol you were looking for):

```
| { Predefined constants, types, procedures, }
| { and functions (such as True, Integer, or }
| { writeln) do not have actual declarations.}
| { Instead they are built into the compiler }
| { and are treated as if they were declared }
| { at the beginning of the System unit.      }
```

Reading the source code of this unit can be rather tedious, also because here you can find some of the lower level code of the entire run-time library. So I've decided to describe only a very limited selection of its content.

Selected System Types

As mentioned above, the `system` unit defines core data types and many types aliases for different numeric types, other ordinal types and strings. There are other core

514 - 17: TObject and the System Unit

data types (comprising enumerations, records, and strong type aliases) used at the low level by the system, that is worth having a look to:

- `TVisibilityClasses` is an enumeration used for RTTI visibility settings (see Chapter 16 for more details)
- `TGUID` is a record used to represent a GUID on Windows, but also on all other supported operating systems
- `TMethod` is a core record representing the structure used for event handler, with a pointer to a method address and one to a current object (mentioned briefly in Chapter 10)
- `TMonitor` is a record implementing a thread synchronization mechanism (called “monitor”) invented by C.A.R Hoare and Per Brinch Hansen and detailed on Wikipedia under the voice “Monitor synchronization”. This is a core threading support feature of the language itself, as the `TMonitor` information is attached to any object in the system.
- `TDateTime` is a strongly typed alias of the `Double` type, used to store date information (in the integral part of the value) and time information (in the decimal part). Further aliases include the types `TDate` and `TTime`. These types were covered in Chapter 2.
- `THandle` is an alias of numeric types, used to represent a reference to an operating system object, generally called a “handle” (as least in the Windows API jargon).
- `TMemoryManagerEx` is a record holding the core memory operations that allows replacing the system memory manager with a custom one (this is the newer version of `TMemoryManager`) still available for backwards compatibility.
- `THeapStatus` is a record with information about the status of the heap memory, shortly mentioned in Chapter 13.
- `TTextLineBreakStyle` is an enumeration indicating the line break style for text files on the given operating system. The `DefaultTextLineBreakStyle` global variable of this type holds the current information, used by many system libraries. Similarly the `sLineBreak` constant expresses the same information as a string value.

Interfaces in the System Unit

There are several interface types (and a few classes implementing interfaces at the core level) that are part of the `System` unit and it is worth looking into. Interfaces

were covered in Chapter 11. Here are the most relevant interface-related types in the `System` unit:

- `IInterface` is the basic interface type all other interfaces inherit from and has the same fundamental role that `TObject` has for classes.
- `IInvokable` and `IDispatch` are interfaces supporting forms of dynamic invocation (partly tied to Windows COM implementation)
- Enumerator support and comparison operations are defined by the following interfaces: `IEnumerator`, `IEnumerable`, `IEnumerator<T>`, `IEnumerable<T>`, `IComparable`, `IComparable<T>`, and `IEquatable<T>`.

There are also a few core classes that offer a basic implementation of interfaces. You often inherit from these classes when implementing an interface, as also covered in Chapter 11:

- `TInterfacedObject`, a class that has a basic implementation of reference counting and interface ID checking
- `TAggregatedObject` and `TContainedObject`, two classes that offer special implementation for aggregated object and the `implements` syntax.

Selected System Routines

The number of intrinsic and standard procedures and functions in the `System` unit is quite large, but most of them are not commonly used. Here is a very limited selection of core functions and procedures every Object Pascal developer should know about:

- `Move` is the core memory copy operation in the system, just copying the given number of bytes from a memory location to another (very powerful, very fast, but a tad dangerous)
- The `ParamCount` and the `ParamStr` functions can be used for processing command line parameters of an application (and do actually work on GUI systems like Windows and Mac as well).
- `Random` and `Randomize` are two classic functions (likely coming from BASIC) providing you with random values (but pseudo-random only if you remember to call `Randomize`, otherwise you get the same sequence at every execution)
- A significant number of core mathematical functions, totally omitted here

516 - 17: TObject and the System Unit

- Many string processing and string conversion functions (between UTF-16 Unicode, UTF-8, ANSI, and other string formats), some of which are platform specific

note Some of these functions have an indirect definition. In other words, the function is actually a pointer to the real function, so that the original system behavior can be dynamically replaced in code at run-time. (If you know what you are doing, of course, as this can be a good way to trash the memory of your application).

Predefined RTTI Attributes

The last group of data types that I want to mention in this chapter relates to attributes, the extra RTTI information you can attach to any symbol of the language. This topic was covered in Chapter 16, but there I didn't mention the predefined attributes in the system.

Here are the attribute classes defined in the `System` unit:

- `TCustomAttributes` is the base class for all custom attributes. This is the base class you have to inherit attributes from (and it is the only way a class is identified by the compiler as being an attribute, as there is no special declaration syntax).
- `WeakAttribute` is used to indicate weak references for interface references (see Chapter 13)
- `UnsafeAttribute` is used to disable reference counting for interface references (also covered in Chapter 13)
- `RefAttribute` is apparently used for reference values.
- `VolatileAttribute` indicates *volatile* variables, which can be modified externally and should not be optimized by the compiler
- `StoredAttribute` is an alternative way to express the stored flag of a property
- `HPPGENAttribute` controls C++ interface file (HPP) generation
- `HFAAttribute` can be used to fine tune ARM 64-bit CPU parameters passing, controlling the Homogeneous Floating-point Aggregates (HFA)

There is more to the `System` unit, but that's for expert developers. I'd rather want to move to the last chapter, where I'm touching the `Classes` unit and some of the RTL capabilities.

18: other core rtl classes

If the `TObject` class and the `System` unit can be considered to all effect as being a structural part of the language, something needed by the compiler itself for building any application, everything else in the runtime library can be considered as optional extensions to the core system.

The RTL has a very large collection of system functions, encompassing the most common standard operations and partially dating back to the Turbo Pascal days, predating the Object Pascal language. Many units of the RTL are collections of functions and routines, including core utilities (`SystemUtils`), mathematical functions (`Math`), string operations (`StringUtils`), date and time processing (`DateUtils`) and many others.

In this book I don't really want to delve into this more traditional part of the RTL, but rather focus on core classes, that are foundations of the visual component libraries used in Object Pascal (VCL and FireMonkey) and also of other subsystems. The `TComponent` class, for example, defines the concept of “component-based” architecture. It is also fundamental for memory management and other base features. The `TPersistent` class is key for streaming component representations.

518 - 18: Other Core RTL Classes

There are many other classes we could look at, as the RTL is extremely large and encompasses the file system, the core threading support, the parallel programming library, string building, many different types of collections and containers classes, core geometrical structures (like points and rectangles), core mathematical structures (like vectors and matrices), and much, much more.

Given the focus of the book is really the Object Pascal language, and not a guide to the libraries, here I'm going to focus only on a few selected classes, chosen either for their key role or because they have been introduced over recent years and are largely ignored by developers.

The Classes Unit

The unit at the foundation of the Object Pascal RTL class library (and also of the visual libraries) is appropriately called `System.Classes`. This unit contains a large collection of mostly assorted classes, without a specific focus. It is worth having a brief look at the important ones, and follow up with an in-depth analysis of the most important.

The Classes in the Classes Unit

So here is the short list (of roughly half of the classes actually defined in the unit):

- `TList` is a core list of pointers which is often adapted to be an untyped list. In general it is recommended to use `TList<T>` instead, as covered in Chapter 14.
- `TInterfaceList` is a thread-safe list of interfaces implementing `IInterfaceList`, worth a second look (but not covered here).
- `TBits` is a very simple class for manipulating individual bits in a number or some other value. It is much higher level than doing bit manipulation with shifts and binary or and and operators.
- `TPersistent` is a fundamental class (the base class of `TComponent`), covered in detail in the next section.
- `TCollectionItem` and `TCollection` are classes used to define collection properties, that is properties with an array of values. These are important classes for component developers (and indirectly when using components), not as much for generic end users code.

- `TStrings` is an abstract list of strings, while `TStringList` is an actual implementation of the base `TStrings` class providing storage for the actual strings. Each item also has an object attached, and these are the standard way to use string lists for name/value string pairs. There is some further information about this class in the section “Using String Lists” at the end of this chapter.
- `TStream` is an abstract class representing any sequence of bytes with sequential access, which can encompass many different storage options (memory, files, strings, sockets, BLOB fields, and many others). The `Classes` unit defines many of the specific stream classes, including `THandleStream`, `TFileStream`, `TCustomMemoryStream`, `TMemoryStream`, `TBytesStream`, `TStringStream`, and `TResourceStream`. Other specific streams are declared in different RTL units. You can read an introduction to streams in the section “Introducing Streams” of this chapter.
- Classes for low-level component streaming, like `TFilter`, `TReader`, `TWriter`, and `TParser`, mostly used by component authors... and not even that often by them.
- `TThread` class, which defines support for platform-independent, multithreaded applications. There is also a class for asynchronous operations, called `TBaseAsyncResult`.
- Classes for the implementation of the observer pattern (used for example in visual live bindings), including `TObservers`, `TLinkObservers`, and `TObserverMapping`.
- Classes for specific custom attributes like `DefaultAttribute`, `NoDefaultAttribute`, `StoredAttribute`, and `ObservableMemberAttribute`.
- The fundamental `TComponent` class, the base class of all visual and non visual components in both VCL and FireMonkey, covered in detail later in this chapter.
- Classes for actions and action lists support (actions are abstraction of “commands” issues by UI elements or internally), including `TBasicAction` and `TBasicActionLink`.
- The class representing a non visual component container, `TDataModule`.
- Higher level interfaces for file and stream operations, including `TTextReader` and `TTextWriter`, `TBinaryReader` and `TBinaryWriter`, `TStringReader` and `TStringWriter`, `TStreamReader` and `TStreamWriter`. These classes are also covered in this chapter.

The TPersistent Class

The `TObject` class has a very important subclass, one of the foundations of the entire library, called `TPersistent`. If you look to the methods of the class, its importance might be surprising... as the class does very little. One of the key elements of the `TPersistent` class is that it is defined with the special compiler option `{M+}`, whose role is to enable the `published` keyword, covered in Chapter 10.

The `published` keyword has a fundamental role for streaming properties, and this explains the name of the class. Originally, only classes inheriting from `TPersistent` could be used as the data type of published properties. The extension of RTTI in later versions of the Object Pascal compiler changed the picture a bit, but the role of the `published` keyword and the `{M+}` compiler option are still there.

note Using today's compiler, if you add the `published` keyword to a class that doesn't inherit from `TPersistent` and hasn't got the `{M+}` compiler flag, the system will add the proper support anyway, indicating that with a warning.

What is the specific role of the `TPersistent` class in the hierarchy? First, it serves as the base class of `TComponent`, which I'll introduce in the next section. Second, it is used as the base class for data types used for property values, so that these properties and their internal structure can be properly streamed. Examples are classes representing list of strings, bitmaps, fonts, and other objects.

If the most relevant feature of the `TPersistent` class is its “activation” of the `published` keyword, it still has a couple of interesting methods worth examining. The first is the `Assign` method, which is used to make a copy of the object data from one instance to another (a deep copy, not a copy of the references). This is a feature each persistent class used for property values should manually implement (as there is no automatic deep copy operation in the language). The second is the reverse operation, `AssignTo`, which is protected. These two methods and the other few available in the class are mostly used by component writers, rather than by application developers.

The TComponent Class

The `TComponent` class is the cornerstone of the components libraries that are most often used in conjunction with Object Pascal compilers. The concept of a component is basically that of a class that has some extra design-time behavior, specific streaming capabilities (so that the design time configuration can be saved and restored in a

running application), and the PME (property-method-event) model we discussed in Chapter 10.

This class defines a significant number of standard behaviors and features, introduces its own memory model based on a concept of objects ownership, cross components notifications, and much more. While not doing a complete analysis of all of the properties and methods, it is certainly worth focusing on some of the key features of the `TComponent` class for its central role in the RTL.

Another critical features of the `TComponent` class is the fact it introduces a virtual `Create` constructor, critical for the ability of creating an object from a class reference whilst still invoking the specific constructor code of the class. We touched on it in Chapter 12, but this is a peculiar feature of the Object Pascal language, worth understanding.

Components Ownership

The ownership mechanism is a key element of the `TComponent` class. If a component is created with an owner component (passed as parameter to its virtual constructor), this owner component becomes responsible for destroying the owned component. In short, each component has a reference to its owner (the `Owner` property), but also a list of components it owns (the `Components` array property) and their number (the `ComponentCount` property).

By default, when you drop a component in a designer (a form, a frame or a data module), this is considered the owner of the component. When you are creating a component in code, it is up to you to specify an owner, or pass `nil` (in which case you'll be responsible for freeing the component from memory yourself).

You can use the `Components` and `ComponentCount` properties to list the components owned by a component (`aComp` in this case), with code like:

```
var
  I: Integer;
begin
  for I := 0 to aComp.ComponentCount - 1 do
    aComp.Components[I].DoSomething;
```

Or use the native enumeration support, by writing:

```
var
  childComp: TComponent;
begin
  for childComp in aComp do
    childComp.DoSomething;
```

When a component is destroyed, it removes itself from the owner list (if any) and it destroys all of the component it owns. This mechanism is crucial to memory man-

agement in Object Pascal: Given there is no garbage collection, ownership can solve most of your memory management issues, as we partially saw in Chapter 13.

As I mentioned, generally all of the components in a form or data module have the form or data module as owner. As long as you free the form or data module, the components they hosts are also destroyed. This is what happens when components are created from a stream.

Components Properties

Beside the core ownership mechanism (which also includes notifications and other features not covered here) any component has two published properties:

- `Name` is a string with the component name. This is used to find a component dynamically (calling the `FindComponent` method of the owner) and to connect the component with the form field referring to the it. All components owned by the same owner must have different names, but their name can also be empty. Two short rules here: set proper component names to improve your code readability and never change the name of a component at runtime (unless you are really aware of the possibly nasty side effects).
- `Tag` is a `NativeInt` value (used to be an `Integer` in the past) not used by the library but available for you to connect extra information to the component. The type is size-compatible with pointers and object references, which are often stored in a component's `Tag`.

Component Streaming

The streaming mechanism used by both FireMonkey and VCL to create FMX or DFM files is based around the `TComponent` class. The Delphi streaming mechanism saves the published properties and events of a component and its sub-components. That's the representation you get in a DFM or FMX file, and also what you get if you copy and paste a component from the designer into a text editor.

There are methods to obtain the same information at run time, including the methods `WriteComponent` and `ReadComponent` of the `TStream` class, but also the `ReadComponentRes` and `WriteComponentRes` methods of the same class, and the `ReadRootComponent` and `WriteRootComponent` of the `TReader` and `TWriter` special classes that help dealing with component streaming. These operations generally use the binary representation of form streams: You can use the global procedure `ObjectResourceToText` to convert the form binary representation to the textual one, and `ObjectTextToResource` for the opposite conversion.

A key element is that the streaming is not a complete set of the published properties of a component. The streaming includes:

- The published properties of a component with a value different from their default value (in other words, default values are not saved to reduce the size)
- Only published properties marked as stored (which is the default). A property with stored set to false (or a function returning false), won't be saved.
- Additional entries not corresponding component properties added at run-time by overriding the `DefineProperties` method.

When a component is created from the stream file, the following sequence occurs:

- The component's virtual `Create` constructor is called (executing the proper initialization code)
- The properties and events are loaded from the stream (in case of events, remapping the method name to the actual method address in memory)
- The `Loaded` virtual method is called to finalize loading (and components can do extra custom processing, this time with the property values already loaded from the stream)

Modern File Access

Borrowing from its ancestor Pascal language, Object Pascal still has keywords and core language mechanisms for processing files. These were basically deprecated when Object Pascal was introduced and I'm not going to touch on them in this book. What I'm going to cover in this section, instead, is a couple of modern techniques for processing files, introducing the `IOUtils` unit, the stream classes, and the readers and writers classes.

The Input/Output Utilities Unit

The `System.IOUtils` unit is a relatively recent addition to the Run Time Library. It defines three records of mostly class methods: `TDirectory`, `TPath`, and `TFile`. While it is quite obvious that `TDirectory` is for browsing folders and finding its files and sub-folders, it might not be so clear what is the difference between a `TPath` and `TFile`. The first, `TPath`, is used for manipulating file name and directory names, with methods for extracting the drive, file name with no path, extension and the

524 - 18: Other Core RTL Classes

like, but also for manipulating UNC paths. The `TFile` record, instead, lets you check the file time stamps and attributes, but also manipulate a file, writing to it or copying it. As usual, it can be worth looking at an example. The `IoFilesInFolder` application project can extract all of the sub-folders of a given folder and it can grab all of the files with a given extension available under that folder.

Extracting Subfolders

The program can fill a list box with the list of the folders under a directory, by using the `GetDirectories` method of the `TDirectory` record, passing as parameter the value `TSearchOption.soAllDirectories`. The result is a string array which you can enumerate:

```
procedure TFormIoFiles.BtnSubfoldersClick(Sender: TObject);  
var  
    PathList: TStringDynArray;  
    StrPath: string;  
begin  
    if TDirectory.Exists (EdBaseFolder.Text) then  
        begin  
            ListBox1.Items.Clear;  
            PathList := TDirectory.GetDirectories(EdBaseFolder.Text,  
                TSearchOption.soAllDirectories, nil);  
            for StrPath in PathList do  
                ListBox1.Items.Add (StrPath);  
        end;  
end;
```

Searching Files

A second button of the program lets you get all of the files of those folders, by scanning each directory with a `GetFiles` call based on a given mask. You can have more complex filtering by passing an anonymous method of type `TFilterPredicate` to an overloaded version of `GetFiles`.

This example uses the simpler mask-based filtering and populates an internal string list. The elements of this string list are then copied to the user interface after removing the full path, keeping only the file name. As you call the `GetDirectories` method you get only the sub-folders, but not the current one. This is why the program searches in the current folder first and then looks into each sub-folder:

```
procedure TFormIoFiles.BtnPasFilesClick(Sender: TObject);  
var  
    PathList, FilesList: TStringDynArray;  
    StrPath, StrFile: string;  
begin  
    if TDirectory.Exists (EdBaseFolder.Text) then  
        begin
```

```

// clean up
ListBox1.Items.Clear;

// search in the given folder
FilesList := TDirectory.GetFiles (EdBaseFolder.Text, '*.pas');
for StrFile in FilesList do
    SFilesList.Add(StrFile);

// search in all subfolders
PathList := TDirectory.GetDirectories(EdBaseFolder.Text,
    TSearchOption.soAllDirectories, nil);
for StrPath in PathList do
begin
    FilesList := TDirectory.GetFiles (StrPath, '*.pas');
    for StrFile in FilesList do
        SFilesList.Add(StrFile);
    end;
end;

// now copy the file names only (no path) to a listbox
for StrFile in SFilesList do
    ListBox1.Items.Add (TPath.GetFileName(StrFile));
end;
end;

```

In the final lines, the `GetFileName` function of `TPath` is used to extract the file name from the full path of the file. The `TPath` record has a few other interesting methods, including a `GetTempFileName`, a `GetRandomFileName`, a method for merging paths, a few to check if they are valid or contain illegal characters, and much more.

Introducing Streams

If the `IOUtils` unit is for finding and manipulating files, when you want to read or write a file (or any other similar sequential access data structure) you can use the `TStream` class and its many descendant classes. The `TStream` abstract class has just a few properties (`Size` and `Position`) along with the basic interface all stream classes share, with the main `Read` and `Write` methods. The concept expressed by this class is sequential access. Every time you read and write a number of bytes, the current position is advanced by that number. For most streams, you can move the position backwards, but there can also be unidirectional streams.

Common Stream Classes

As I mentioned earlier, the `Classes` unit defines several concrete stream classes... includes the following ones:

- `THandleStream` defines a disk file stream referenced with a file handle.

526 - 18: Other Core RTL Classes

- `TFileStream` defines a disk file stream referenced by a file name.
- `TBufferedFileStream` is an optimized disk file stream which uses a memory buffer for extra performance. This stream class has been introduced in Delphi 10.1 Berlin.
- `TMemoryStream` defines a stream of data in memory, that you can also access using a pointer.
- `TBytesStream` represents a stream of bytes in memory, that you can also access like an array of bytes
- `TStringStream` associates a stream to a string in memory.
- `TResourceStream` defines a stream that can read resource data linked into the executable file of an application.

Using Streams

Creating and using a stream can be as simple as creating a variable of the specific type and calling a component's methods to load content from the file. For example, given a stream and a memo component you can write:

```
AStream := TFileStream.Create (FileName, fmOpenRead);  
Memo1.Lines.LoadFromStream (AStream);
```

As you can see in this code, the `Create` method for file streams has two parameters: the name of the file and some flag indicating the requested access mode. As I mentioned streams support read and write operations, but these are rather low level, so I'd rather recommend using the readers and writers classes discussed in the next section. What a direct use of stream provides is comprehensive operations, like loading an entire stream in the code snippet above, or copying one into another:

```
procedure CopyFile (SourceName, TargetName: String);  
var  
    Stream1, Stream2: TFileStream;  
begin  
    Stream1 := TFileStream.Create (SourceName, fmOpenRead);  
    try  
        Stream2 := TFileStream.Create (TargetName,  
            fmOpenWrite or fmCreate);  
        try  
            Stream2.CopyFrom (Stream1, Stream1.Size);  
        finally  
            Stream2.Free;  
        end  
    finally  
        Stream1.Free;  
    end  
end;
```

Using Readers and Writers

A very nice approach for writing to and reading from streams is to use the reader and writer classes that are part of the RTL. There are six reading and writing classes, defined in the `Classes` unit:

- `TStringReader` and `TStringWriter` work on a string in memory (directly or using a `TStringBuilder`)
- `TStreamReader` and `TStreamWriter` work on a generic stream (a file stream, a memory stream, and more)
- `TBinaryReader` and `TBinaryWriter` work on binary data rather than text.

Each of the text *readers* implements a few basic reading techniques:

```
function Read: Integer; overload;
function ReadLine: string;
function ReadToEnd: string;
```

Each of the text *writers* has two sets of overloaded operations without (`write`) and with (`writeLine`) an end-of-line separator. Here is the first set:

```
procedure Write(Value: Boolean); overload;
procedure Write(Value: Char); overload;
procedure Write(const Value: TCharArray); overload;
procedure Write(Value: Double); overload;
procedure Write(Value: Integer); overload;
procedure Write(Value: Int64); overload;
procedure Write(Value: TObject); overload;
procedure Write(Value: Single); overload;
procedure Write(const Value: string); overload;
procedure Write(Value: Cardinal); overload;
procedure Write(Value: UInt64); overload;
procedure Write(const Format: string; Args: array of const); overload;
procedure Write(Value: TCharArray; Index, Count: Integer); overload;
```

Text Readers and Writers

For writing to a stream, the `TStreamWriter` class uses a stream or creates one using the file name, an append/create attribute, and the Unicode encoding passed as parameters.

So we can write, as I did in the `ReaderWriter` application project:

```
var
  Sw: TStreamWriter;
begin
  Sw := TStreamWriter.Create('test.txt',
    False, TEncoding.UTF8);
  try
    Sw.WriteLine('Hello, world');
```


528 - 18: Other Core RTL Classes

```
Sw.WriteLine ('Have a nice day');  
Sw.WriteLine (Left);  
finally  
Sw.Free;  
end;
```

For reading the `TStreamReader`, you can work again on a stream or a file (in which case it can detect the encoding from the UTF BOM marker):

```
var  
SR: TStreamReader;  
begin  
SR := TStreamReader.Create('test.txt', True);  
try  
  while not SR.EndOfStream do  
    Memo1.Lines.Add (SR.ReadLine);  
finally  
  SR.Free;  
end;
```

Notice how you can check for the `EndOfStream` status. Compared to a direct use of text streams (or even strings), these classes are particularly handy to use, and provide good performance.

Binary Reader and Writer

The classes `TBinaryReader` and `TBinaryWriter` are meant for managing binary data rather than text files. These classes generally encapsulate a stream (a file stream or any type of in-memory stream, including sockets and database tables BLOB fields) and have overloaded `Read` and `Write` methods.

As a (rather simple) example I've written the `BinaryFiles` application project. In its first part this program writes a couple of binary elements to a file (the value of a property and the current time) and reads them back, assigning the property value:

```
procedure TFormBinary.BtnWriteClick(Sender: TObject);  
var  
  BW: TBinaryWriter;  
begin  
  BW := TBinaryWriter.Create('test.data', False);  
try  
  BW.Write(Left);  
  BW.Write(Now);  
  Log ('File size: ' + IntToStr (BW.BaseStream.Size));  
finally  
  BW.Free;  
end;  
end;  
  
procedure TFormBinary.BtnReadClick(Sender: TObject);  
var  
  br: TBinaryReader;
```

```

    time: TDateTime;
begin
    br := TBinaryReader.Create('test.data');
    try
        Left := br.ReadInt32;
        Log ('Left read: ' + IntToStr (Left));
        time := br.ReadDouble;
        Log ('Time read: ' + TimeToStr (time));
    finally
        br.Free;
    end;
end;

```

The key rule in using these reader and writer classes is that you have to read the data in the same order you wrote it, or else you'll totally mess up the data. In fact, only the binary data of individual fields is saved, with no information about the field itself. Nothing prevents you from interposing data and metadata in the file, like saving the size of the next data structure before the actual value or a token referring the field.

Building Strings and String Lists

After having a look at files and streams, I want to spend a little time focusing on ways of manipulating string and lists of strings. These are very common operations and there is a rich set of RTL features focused on them. Here I'm only going to introduce a few.

The TStringBuilder class

I have already mentioned in Chapter 6, that unlike other languages, Object Pascal has full support for direct string concatenation, which is actually a rather fast operation. The language RTL, however, also includes a specific class for assembling a string out of fragments of different data types, called `TStringBuilder`.

As a simple example of the use of the `TStringBuilder` class, consider the following code snippet:

```

var
    SBuilder: TStringBuilder;
    Str1: string;
begin
    SBuilder := TStringBuilder.Create;
    SBuilder.Append(12);

```

530 - 18: Other Core RTL Classes

```
SBuilder.Append('hello');  
Str1 := SBuilder.ToString;  
SBuilder.Free;  
end;
```

Notice that we have to create and destroy this `TStringBuilder` object. Another element you can notice above is that there are many different data types that you can pass as parameters to the `Append` function.

Other interesting methods of the `TStringBuilder` class include an `AppendFormat` (with an internal call to `Format`) and an `AppendLine` that adds the `sLineBreak` value. Along with `Append`, there is a corresponding series of `Insert` overloaded methods, as well as a `Remove` and a few `Replace` methods.

note The `TStringBuilder` class has a nice interface and offers good usability. In terms of performance, though, using standard string concatenation and formatting functions can provide better results, unlike other programming languages that define immutable strings and have very bad performance in case of pure string concatenation.

Method Chaining in StringBuilder

A very specific feature of the `TStringBuilder` class is that most methods are functions that return the current object they have been applied to.

This coding idiom opens up the possibility of method chaining, that is calling a method on the object returned by the previous one. Instead of writing:

```
SBuilder.Append(12);  
SBuilder.AppendLine;  
SBuilder.Append('hello');
```

you can write:

```
SBuilder.Append(12).AppendLine.Append('hello');
```

which can also be formatted as:

```
SBuilder.  
  Append(12).  
  AppendLine.  
  Append('hello');
```

I tend to like this syntax better than the original one, but I know it is just syntactic sugar and some people do prefer the original version with the object spelled out on each line. In any case, keep in mind that the various calls to `Append` don't return new objects (so no potential memory leaks), but the exact same object to which you are applying the methods.

Using String Lists

Lists of strings are a very common abstraction used by many visual components, but are also used as a way to manipulate text made of separate lines. There are two main classes for processing lists of strings:

- `TStrings` is an abstract class to represent all forms of string lists, regardless of their storage implementations. This class defines an abstract list of strings. For this reason, `TStrings` objects are used only as properties of components capable of storing the strings themselves.
- `TStringList`, a subclass of `TStrings`, defines a list of strings with its own storage. You can use this class to define a list of strings in a program.

The two classes of lists of strings also have ready-to-use methods to store or load their contents to or from a text file, `SaveToFile` and `LoadFromFile` (which are fully Unicode enabled). To loop through a list, you can use a simple `for` statement based on its index, as if the list were an array, or a `for-in` enumerator.

The Run-Time Library is Quite Large

There is a lot more to the RTL that you can use along with Object Pascal compilers, encompassing a lot of core features for development on multiple operating systems. Covering the entire Run-Time Library in detail would easily fill another book of the same size as this one.

If we consider only the main portion of the library, that is the “*System*” namespace, it includes the following units (from which I’ve removed a few rarely used ones):

- `System.Actions` includes the core support for the actions architecture, which provides a way to represent user commands connected, but abstracted, from the user interface layer.
- `System.AnsiStrings` has the old functions for processing Ansi strings (only on Windows), covered in Chapter 6.
- `System.Character` has the intrinsic type helpers for Unicode characters (the `Char` type), already covered in Chapter 3.
- `System.Classes` provides core system classes and is the unit I covered in detail in the first part of this chapter.

532 - 18: Other Core RTL Classes

- `System.Contnrs` includes the old, non generic, container classes like objects list, dictionary, queue, and stack. I recommend using the generic version of the same classes, when possible.
- `System.ConvUtils` has a library of conversion utilities for different measurement units
- `System.DateUtils` has functions for processing date and time values
- `System.Devices` interfaces with system devices (like GPS, an accelerometer, and so on).
- `System.Diagnostics` defines a record structure for precise measurement of elapsed time in testing code, which I've occasionally used in the book.
- `System.Generics` has actually two separate units, one for generic collections and one for generic types. These units are covered in Chapter 14.
- `System.Hash` has the core support for defining hash values.
- `System.ImageList` includes an abstract, library independent implementation for managing lists of images and portions of a single image as a collection of elements.
- `System.IniFiles` defines an interface for processing INI configuration files, often found in Windows.
- `System.IOUtils` defines records for file system access (files, folders, paths), which were covered earlier in this chapter.
- `System.JSON` includes same core classes for processing data in the commonly used JavaScript Object Notation, or JSON.
- `System.Math` defines functions for mathematical operations, including trigonometric and financial functions. It also has other units in its namespace for vectors and matrixes.
- `System.Messaging` has shared code for messages handling on different operating systems.
- `System.NetEncoding` includes processing for some common Internet encodings, like base64, HTML, and URL.
- `System.RegularExpressions` defines regular expression (*regex*) support.
- `System.Rtti` has the entire set of RTTI classes, as explained in Chapter 16.
- `System.StrUtils` has the core and traditional string processing functions.
- `System.SyncObjs` defines a few classes for synchronizing multithreaded applications.

- `System.SysUtils` has the basic collection of system utilities, with some of the most traditional ones dating back to the early days of the compiler.
- `System.Threading` includes the interfaces, records and classes of the fairly recent Parallel Programming Library.
- `System.Types` has some core additional data types, like `TPoint`, `TRectangle`, and `TSize` records, the `TBitConverter` class, and many more basic data types used by the RTL.
- `System.TypeInfo` defines the older RTTI interface, also introduced in Chapter 16, basically superseded by those in the `System.RTTI` unit.
- `System.Variants` and `System.VarUtils` have functions for working with variants (a language feature covered in Chapter 5).
- `System.Zip` interfaces a file compression and decompression library.

There are also several other portions of the RTL that are sub-sections of the *System* name space, with each section encompassing multiple units (occasionally a large numbers, like the `System.Win` namespace), including HTTP clients (`System.Net`), and Internet of Things support (`System.Beacon`, `System.Bluetooth`, `System.Sensors`, and `System.Tether`). There are also, of course, translated APIs and headers file for interfacing with all of the supported operating systems.

Again, there is a wealth of ready to use RTL functions, types, records, interfaces, and classes, that are there for you to explore, to leverage the power of the Object Pascal. Take your time browsing the system documentation to learn more too.

534 - 18: Other Core RTL Classes

in closing

Chapter 18 marks the end of the book, save for the following three appendixes. This was originally my first book focused exclusively on the Object Pascal language, and I'm doing my best effort to keep updating it and maintaining the book text and the examples over time. I did a PDF only update for Delphi 10.1 Berlin and now this new version you are reading for Delphi 10.4 Sydney.

Again refer to the Introduction for getting the latest book source code from GitHub and visit the book web site or my blog for future information and updates.

I hope you've enjoyed reading the book as much as I've liked writing it and writing about Delphi over the last 25 years. Happy coding with Delphi!

end.

This final section of the book has a few appendices, that focus on specific side issues worth considering, but out of the flow of the text. There is a short history of the Pascal and Object Pascal languages and a glossary.

Appendix Summary

Appendix A: The Evolution of Object Pascal

Appendix B: Glossary of Terms

Appendix C: Index

538 - end.

a: the evolution of object pascal

Object Pascal is a language built for the growing range of today's computing devices, from smart phones and tablets to desktops and servers. It didn't just appear out of thin air. It has been carefully designed on a solid foundation to be the tool of choice for modern programmers. It provides an almost ideal balance between the speed of programming and the speed of the resulting programs, clarity of syntax and power of expression.

The foundation that Object Pascal is built upon is the Pascal family of programming languages. In the same way that Google's Go language or Apple's Objective-C language are rooted in C, Object Pascal is rooted in Pascal. No doubt you would have guessed that from the name.

This short appendix includes a brief history of the family of languages and actual tools around Pascal, Turbo Pascal, Delphi's Pascal, and Object Pascal. While it is not really necessary to read this to learn the language, it is certainly worth understanding the language's evolution and where it is today.

The Object Pascal programming language we use today in Embarcadero development tools was invented in 1995 when Borland introduced Delphi, which at the time

540 - A: The Evolution of Object Pascal

was its new visual development environment. The first Object Pascal language was extended from the language already in use in the Turbo Pascal products, where the language was generally referenced as Turbo Pascal. Borland didn't invent Pascal, it only helped make it very popular, and extend its foundations to overcome some of its limitations compared to the C language.

The following sections cover the history of the language from Wirth's Pascal to the most recent LLVM-based Delphi's Object Pascal compiler for ARM chips and mobile devices.

Wirth's Pascal

The Pascal language was originally designed in 1971 by Niklaus Wirth, professor at the Polytechnic of Zurich, Switzerland. The most complete biography of Wirth is available at <http://www.cs.inf.ethz.ch/~wirth>.

Pascal was designed as a simplified version of the Algol language for educational purposes. Algol itself was created in 1960. When Pascal was invented, many programming languages existed, but only a few were in widespread use: FORTRAN, Assembler, COBOL, and BASIC. The key idea of the new language was order, managed through a strong concept of data types, declaration of variables, and structured program controls. The language was also designed to be a teaching tool, that is to teach programming using best practices.

Needless to say that the core tenets of Wirth's Pascal have had a huge influence on the history of all programming languages, well beyond and above those still based on the Pascal syntax. As for teaching languages, too often schools and universities have followed other criteria (like job requests or donations from tool vendors) rather than looking at which language helps learning the key concepts of programming better. But that is another story.

Turbo Pascal

Borland's world-famous Pascal compiler, called Turbo Pascal, was introduced in 1983, implementing "Pascal User Manual and Report" by Jensen and Wirth. The Turbo Pascal compiler has been one of the best-selling series of compilers of all time, and made the language particularly popular on the PC platform, thanks to its

balance of simplicity, power, and price. The original author was Anders Hejlsberg, later father of the very popular C# and TypeScript programming languages at Microsoft.

Turbo Pascal introduced an Integrated Development Environment (IDE) where you could edit the code (in a WordStar compatible editor), run the compiler, see the errors, and jump back to the lines containing those errors. It sounds trivial now, but previously you had to quit the editor, return to DOS; run the command-line compiler, write down the error lines, open the editor and jump to the error lines.

Moreover Borland sold Turbo Pascal for 49 dollars, where Microsoft's Pascal compiler was sold for a few hundred. Turbo Pascal's many years of success contributed to Microsoft eventually dropping its Pascal compiler product.

You can actually download a copy of the original version of Borland's Turbo Pascal from the *Museum* section of the Embarcadero Developer Network:

|| <http://edn.embarcadero.com/museum>

history After the original Pascal language, Nicklaus Wirth designed the Modula-2 language, an extension of Pascal syntax now almost forgotten, which introduced a concept of modularization very similar to the concept of units in early Turbo Pascal and today's Object Pascal.

A further extension of Modula-2 was Modula-3, which had object-oriented features similar to Object Pascal. Modula-3 was even less used than Modula-2, with most commercial Pascal language development moving towards Borland and Apple compilers, until Apple abandoned Object Pascal for Objective-C, leaving Borland with almost a monopoly on the language.

The early days of Delphi's Object Pascal

After 9 versions of Turbo and Borland Pascal compilers, which gradually extended the language into the Object Oriented Programming (OOP) realm, Borland released Delphi in 1995, turning Pascal into a visual programming language. Delphi extended the Pascal language in a number of ways, including many object-oriented extensions which are different from other flavors of Object Pascal, including those in the *Borland Pascal with Objects* compiler (the last incarnation of Turbo Pascal).

history Year 1995 was really a special year for programming languages, as it saw the debut of Delphi's Object Pascal, Java, JavaScript, and PHP. These are some of the most popular programming languages still in use today. In fact, most other popular languages (C, C++, Objective-C, and COBOL) are much older, while the only newer popular language is C#. For a history of programming languages you can see http://en.wikipedia.org/wiki/History_of_programming_languages.

With Delphi 2, Borland brought the Pascal compiler to the 32-bit world, actually re-engineering it to provide a code generator common with the C++ compiler. This brought many optimizations previously found only in C/C++ compilers to the Pascal language. In Delphi 3 Borland added to the language the concept of interfaces, making a leap forward in the expressiveness of classes and their relationships.

With the release of version 7 of Delphi, Borland formally started to call the Object Pascal language the Delphi language, but nothing really changed in the language at that time. At that time Borland also created Kylix, a Delphi version for Linux, and later created a Delphi compiler for Microsoft .NET framework (the product was Delphi 8). Both projects were later abandoned, but Delphi 8 (released at the end of 2003) marked a very extensive set of changes to the language, changes that were later adopted in the Win32 Delphi compiler and all other following compilers.

Object Pascal From CodeGear to Embarcadero

With Borland unsure about its investments in development tools, later versions like Delphi 2007, were produced by CodeGear, a subsidiary of the main company. This subsidiary (or business unit) was later sold to Embarcadero Technologies. After that release, the company re-focused on growing and extending the Object Pascal language, adding long-awaited features like Unicode support (in Delphi 2009), generics, anonymous methods or closures, extended runtime type information or reflection, and many other significant language features (mostly covered in Part III of this book).

At the same time, alongside the Win32 compiler the company introduced a Win64 compiler (in Delphi XE2) and a macOS compiler, getting back to a multi-platform strategy after the attempt done earlier on Linux with the short-lived Kylix product. This time however the idea was to have a single Windows development environment and cross-compile to other platforms. The Mac support was only the beginning of the company's multi-device strategy, embracing desktop and mobile

platforms, like iOS and Android. This strategy was made possible by the adoption of a new GUI framework, called FireMonkey.

Going Mobile

The shift to mobile and the first Object Pascal compiler for ARM chips (as all previous platforms Delphi supported were only on Intel x86 chips) have been tied to an overall re-architecture of the compilers and the related tools (or “compiler toolchain”) based on the open LLVM compiler architecture.

note LLVM is the short name of the LLVM Compiler Infrastructure or “*a collection of modular and reusable compiler and toolchain technologies*,” as you can read at <https://llvm.org/>

The ARM compiler for iOS released in Delphi XE4 was the first Object Pascal compiler based on LLVM, but also the first to introduce some new features like Automatic Reference Counting (or ARC, now removed from the language).

Later in the same year (2013), Delphi XE5 added support for the Android platform, with a second ARM compiler based on LLVM. To summarize, Delphi XE5, shipped with 6 compilers for the Object Pascal language (for the Win32, Win64, macOS, iOS Simulator on Mac, iOS ARM, and Android ARM support). All these compilers support a largely common language definition, with a few significant differences I’ve covered in detail throughout the book.

In the first few months of 2014, Embarcadero released a new development tool based on the same core mobile technologies and called Appmethod. In April 2014, the company also released the XE6 version of Delphi, while September 2014 saw the third release of AppMethod and Delphi XE7, followed in spring 2015 by Delphi XE8, which included the first ARM 64-bit compiler, targeting iOS.

The Delphi 10.x Period

After Delphi 10 Seattle, with the acquisition of the company by Idera Corp., Embarcadero created the 10.x series of releases: Delphi 10.1 Berlin, Delphi 10.2 Tokyo, Delphi 10.3 Rio, and Delphi 10.4 Sydney. Over these versions Embarcadero added support for new target platforms and operating systems: Linux 64-bit, Android 64-bit and macOS 64-bit. The company also refocused on the Windows VCL library by adding specific support for the Windows 10 operating system.

Over the course of the 10.x series Embarcadero continued to evolve the Object Pascal language with the introduction of features like inline variables declarations and custom managed records, plus many other smaller enhancements, all covered in this book.

b: glossary

A

Abstract Class	A class that is not fully defined and provides only the interface of the method that subclasses should implement.
Ambiguous call	This is the error message you receive in case the compiler has two or more options to resolve a function call and has no way to determine automatically which one you are trying to call.
Android	The name of Google's operating system for phones and tablets, embraced by hundreds of hardware vendors (beside Google) given its open nature. Android is currently the most used operating system of the world, having surpassed Microsoft Windows.
Anonymous Method	<p>An anonymous method or anonymous function is a function that is not associated with a function name and can be assigned to a variable or passed as an argument to another function, which can later execute its code.</p> <p>You might think that anonymous methods are a little</p>

magical compared with regular functions. Well they are! The real magic is that they can access variables from the block in which they are declared even if they finally run in a different block.

The anonymous function and the variables it can access are known as a *closure*, which is another name used for the same feature.

API

An Application Programming Interface (API) is provided by software (like operating systems) so that application programs can work with it. For example, when an application displays a line of text on a screen, typically it calls a function in the computer's GUI. The collection of functions provided by the computer's GUI is known as the GUI's API.

Generally when software provides an API for the language, it is written in. For example, the Microsoft Windows provides an API geared towards the C and C++ languages.

Note: The Object Pascal `WinAPI.Windows` unit provides an Object Pascal API to Microsoft Windows removing the hassle of directly calling functions written to be called from C or C++.

B

Boolean Expression

A Boolean expression is an expression that evaluates to either true or false. A simple example being `1 = 2` which happens to be false. The Boolean expression does not have to be a traditional mathematical expression, it could simply be a variable of the Boolean type or even a call to a function which returns a Boolean value.

C

Cardinal	A Cardinal number is one of the natural numbers. Simply put that means a number that can be used to count things and that is always greater than or equal to zero.
Class	<p>A class is a definition of the properties and methods and data fields that an object (of that class) will have when it is created.</p> <p><i>Note:</i> Not all object oriented languages require classes to define objects. Objects in JavaScript, IO and Rebol can be defined directly without first defining a class.</p> <p><i>Note:</i> The definition of a record is very similar to the definition of a class in Object Pascal. A record has members which fulfill the same function as properties do for a class and procedures and functions which do what methods do for a class.</p>
Code Point	The numeric value of an element of the Unicode character set. Each letter, number, punctuation of each alphabet of the world has a Unicode code point representing it.
Compiler Directive	A compiler directive is a special instruction to the compiler, that alters its standard behavior. Compiler directives are assigned with special words prefixed by the \$ sign, or can be set in the Project Options.
Components	<p>Components are prebuilt, ready-to-use code objects that can be easily combined with both application code and other components to dramatically reduce the time taken to develop applications.</p> <p>The VCL library and the FireMonkey Platform are two large collections of such components supplied with Delphi.</p>
COM	Component Object Model is a core part of the Microsoft Windows architecture.

548 - B: Glossary

Control	A control is an element of a GUI such as a button, a text entry field, an image container, etc. Controls are often indicated as visual components.
CPU	The CPU or Central Processing Unit is the core of any computer and what actually executes the code. The Object Pascal language statements need to be translated to assembly code to be understood by the CPU. Notice you have a CPU view in the debugger, not something for the newcomers. The CPU often works alongside an FPU.

D

Data Type	A data type indicates the storage requirement and the operations you can perform on a variable of that type. In Object Pascal, each variable has a specific data type, as it happens for strongly-typed programming languages.
Design Patterns	<p>Looking at software architectures that different developers use to solve different problems, you can notice similarities and common elements. A design pattern is the acknowledgment of such a common design, expressed in a standard way, and abstracted enough to be applicable in a number of different situations. The design patterns movement in the software world started in 1994 when Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides wrote the book “Design Patterns, Elements of Reusable Object-Oriented Software” (Addison-Wesley, 1994, ISBN: 0-201-633612). The authors are often indicated as “Gamma et al.”, but more frequently as the “Gang of Four” or simply “GoF”. The book is often colloquially referenced as the “GoF book”.</p> <p>In the GoF book the authors describe the notion of software patterns, indicate a precise way of describing them, and provide a catalog of 23 patterns, divided in three groups: creational, structural, and behavioral.</p>

DLL A Dynamic Link Library is a library of functions that are not included in an application's executable code but stored in a separate file. Instead, when the application runs, it loads the library into memory and is then able to call the functions contained in the library. These libraries are normally designed to be used by many applications. On platforms other than Windows, the same type of library is called a Shared Object (or SO).

E

Event An event is an action or operation happening in an application, such as a click with the mouse or a form resize. Delphi implements events via a special property of a class that allows an object to delegate some “behavior” to an external method. Events are part of the RAD development model.

F

FireMonkey FireMonkey or FMX is a library of visual and non-visual components supplied with Delphi. The components are cross-platform, so they'll work equally well on Windows, macOS, iOS, Android, and even Linux (via the FMXLinux add-on library).

Form Form is the term used for a window in the VCL and FireMonkey libraries.

File System A file system is part of a computer's operating system that organizes how data is stored on the computer and manages data storage and retrieval.

FPU The FPU or Floating Point Unit is a companion to the CPU focused on executing complex floating point number calculations extremely fast.

550 - B: Glossary

Function	A function is a block of code that performs some action (or computation) and returns a result. It can accept a pre-specified number of parameters to vary the computation.
Function Overloading	Function overloading is a feature of programming languages that are strict about variable types that allows a programmer to declare different versions of a Function that can accept different types of parameter.

G

Global Memory	Global memory is a static memory area for global variables of your applications. This memory is used for the entire lifetime of an application, and it cannot grow (see Heap memory for dynamic allocated memory area). Global memory is used sparingly in Object Pascal applications.
GUI	A Graphical User Interface that allows users to interact with computers, tablets and phones through graphical icons and other visual indicators. Most user interaction with a GUI is performed by pointing, touching, pressing, swiping and other gestures using a mouse (or similar pointing device) or fingers.

H

Heap Memory	The heap is a memory area for dynamically allocated memory blocks. As the name implies, there is no structure or sequence in heap memory allocation. Whenever a block is needed it is taken from a free area. Lifetime of individual blocks is different, and order of allocation and de-allocation are not related. The heap memory is used for the data of objects, strings, dynamic arrays and other reference types
-------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

(See References), but also for manually allocated blocks (see Pointers). The heap is large but not infinite, and if you don't release unused objects from memory, your application will eventually run out of memory.

I

IDE

An *Integrated Development Environment* is a single application that provides a developer with a wide range of tools so that they can be highly productive. As a minimum an IDE will provide a source code editor, build automation tools and a debugger. The modern idea of an IDE was invented along with the first few Turbo Pascal compilers than came from Borland, the precursors of today's Object Pascal IDEs by Embarcadero Technologies.

The Object Pascal IDE supplied with Delphi is very sophisticated and includes, for example, GUI design, code templates, code refactoring and integrated unit testing.

(Type) Inheritance

Type inheritance is one of the core tenets of Object Oriented Programming (OOP). The idea is that a data type can extend an existing data type, adding new features to it. This type extension is known as type inheritance, along with terms like base and descendant class, or parent and child class.

Interface

Generally refers to an abstract declaration of what a software module can do. In Object Pascal an interface is a purely abstract class definition (made only of methods, and with no data), like in C# or Java. See Chapter 11 for full coverage.

However the language also still has the concept of interface for a unit, in which case this is the section of the unit that declares what it visible to other units. The same `interface` keyword is used in both cases.

iOS	The name of the operating system powering Apple's iPhones, iPads, and similar devices.
-----	----------------------------------------------------------------------------------------

M

Method	A method is a function or procedure that is tied to an object. Methods have access for all the data stored in the object.
--------	---------------------------------------------------------------------------------------------------------------------------

O

Object	An object is a combination of some data items (properties and fields) and code (methods). An object is an instance of a class, which defines a family (or type) of objects.
OOP	Object Oriented Programming is the conceptual structure behind Object Pascal, based on concepts like classes, inheritance, and polymorphism. Modern Object Pascal supports also other programming paradigms, thanks to features like generics, anonymous methods and reflection.
Ordinal Type	An ordinal type is data type made of elements that can be counted and have a sequence. You can think of integer numbers, but characters also have a sequence, and even custom enumerated types.
macOS	The old name of the operating system of Apple Mac computers, now replaces by Mac OS

P

Pointer	A pointer is a variable holding directly a memory address. A pointer can refer to the location of some data or of a function in memory. Pointers are not commonly used, while references (see Reference) are opaque and managed pointers that are extremely common, but also significantly easier to use.
Polymorphism	Polymorphism is the ability for a call to a method to assume “different forms” (that is, end up doing different operations) depending on the object is is applied to. It is a standard trait of OOP languages.
Procedure	A procedure is a block of code (or sub-program) that can be called from other parts of a program. A procedure can accept parameters to vary what it does. Differently from a function, a procedure doesn't return a value.
Project Options	A set of configuration options that affect the overall structure of an application project, but also how the compiler and linker behave.
Property	A property defines the state of an object, abstracting from the actual implementation, given a property can be mapped to data or use methods to read and write the value.

R

RAD	Rapid Application Development is a characteristic of a development environment that make it easy and fast to build applications. RAD tools are generally based on visual designers, although this is a rather old definition seldom used today.
Record	A simple record is a collection of data items that are stored in a structured way. Records are defined in a

	<p>type definition showing the order and type of the individual data items in the record.</p> <p>Object Pascal also includes advanced records which can have methods similarly to an object.</p>
Recursion	<p>Recursion or recursive call is a way to describe a function that keep calling itself until a given condition is met. A recursive call is often a better alternative to a loop or cycle. An example of a recursive implementation of a multiplication, would be to take the value of the first number, and add to it the same number multiplied by the other minus one, until the other numbers becomes zero.</p>
Reference	<p>A reference is a variable that refers to some data elsewhere in memory, rather than storing it directly. In Object Pascal variable of types like classes and string, but also interfaces and dynamic arrays, are references. Differently from pointers (see Pointers) references are generally managed by the compiler and runtime library and require little low-level knowledge and direct memory manipulation by the developer.</p>
RTTI (or Reflection)	<p>An acronym of Run Time Type Information, is the ability to access type information (traditionally only available to compilers) in the actual application at run time. Other programming environments refer to this feature as reflection.</p>
Run-Time Library (RTL)	<p>This is a collection of pre-written routines that the compiler automatically includes with application code to build the executable application. It includes support for many fundamental operations, especially those requiring interaction with the operating system when the application is run (e.g. allocating memory, reading and writing data, interacting with the file system).</p>

S

SDK	A Software Development Kit is a set of software tools with which to build software for a specific environment. Each operating system provides an SDK including the API libraries (Application Programming Interfaces) and developer tools necessary to build, test, and debug applications for the platform.
Search Path	A set of folders the compiler will search when looking for an external unit referenced in a uses statement
Stack (memory)	<p>The stack is a dynamically and orderly allocated memory area. Every time you call a method, a procedure, or a function, this reserves its own memory area (for local variables, including temporary ones, and parameters). As the method returns, the memory is cleared, in a very orderly fashion. The only real scenario for running out of the stack memory is when a method enters an infinite recursive call (see Recursion).</p> <p><i>Note:</i> In most cases, local variables allocated on the stack are not initialized to zero: you need to set their value before using them.</p>

U

Unicode	Unicode is a standard way of recording individual text characters as binary data (a sequence of 0s and 1s). Text can be reliably exchanged between programs, processed and displayed if it conforms to the Unicode standard. The standard is very large covering more than 110,000 different characters from around 100 different writing alphabets and scripts.
---------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

V

VCL	The Visual Component Library is a massive set of Visual Components supplied with Delphi. The GUI components of the VCL are native Windows GUI components.
Virtual Methods	A Virtual Method is a function or procedure declared in the type definition of a class that can be overridden by classes which are its sub-classes. The so-called base class can also include a definition for the method that can be used by the sub-classes. If the base class doesn't define a default version of the method, any sub-class must provide a definition of the Virtual Method.

W

Window	A window is an area of the screen that contains GUI elements with which a user can interact. A GUI application can display multiple windows. In VCL and FireMonkey, windows are defined using a Form object.
Windows	The name of Microsoft ubiquitous operating system, which pioneered (along with other operating systems of the time like Apple Mac operating system) the concept of graphical windows (see entry above).

c: index

1

1252 Code Page.....	175
1900.....	83
1983.....	540
1995.....	3, 541

6

64-bit.....	63, 71, 169, 408, 543
-------------	-----------------------

A

Abstract.....	36, 255
Abstract Classes.....	322, 545
Access Violations.....	391
ActionList.....	316
Adapter Pattern.....	339
Additive Operators.....	81
Address Of.....	148, 152, 168
AfterDestruction.....	278
AJAX.....	460
Algol.....	54, 540
Algorithms + Data Structures = Programs....	89
Alignment.....	147, 234
Allen Bauer.....	6, 129

Alphabet.....	174
Ambiguous Calls.....	118, 545
And.....	81
And.....	82
Anders Hejlsberg.....	211, 251, 541
Android.....	49, 543, 545
Android.....	549
Angle Brackets.....	399, 402
Anonymous Delegate.....	446
Anonymous Event Handlers.....	454
Anonymous Methods.....	446, 545
ANSI.....	205
AnsiChar.....	67, 200
ANSIString.....	206p.
API.....	546, 555
Append.....	530
Apple.....	541, 552, 556
Apple's Instruments Tool.....	387
Application.....	277
Appmethod.....	543
ArcCosh.....	72
ArcExperiments.....	
Example.....	378
ArcExperiments Example.....	378
ARM Chips.....	543

558 - C: Index

Array.....35
Array Properties.....294, 318
Arrays.....132
Arrays Of Records.....145
As.....81, 259, 325, 337, 391
ASCII.....174
Assembler.....540
Assign.....236, 372, 520
Assigned.....170, 392
Assignments.....52, 54
Attribute Classes.....482
Attributes.....481
AutoSize.....234

B

Banker's Rounding.....87
Barry Kelly.....478, 487
BASIC.....30, 51, 540
Basic Multilingual Plane.....177
BeforeConstruction.....278
Begin.....34
Bertrand Meyer.....239
Bitwise Operators.....81
BOM Marker.....528
Boolean.....62, 67
Boolean Expression.....546
Boolean Type.....546
BoolToStr.....67
Borland.....540p.
 Borland C++ 4.0 Object-Oriented
 Programming.....397
Break.....102p., 113
Buffer Overruns.....384
Button1Click.....23
Byte.....62
Byte Order Mark.....178
ByteLength.....202

C

C...29, 31, 54, 63, 65, 67, 81, 84, 90p., 94p., 99,
 105, 107, 114, 125, 132, 150
C#. 25, 29, 51, 54, 73, 81, 94, 106, 148, 211, 214,
 216, 226, 238, 243, 248, 251, 257, 259, 266,
 286p., 294, 297, 301pp., 321, 341, 397, 415,
 446

C++....25, 29, 43, 51, 65, 106, 114, 132, 153, 211,
 221, 226, 232, 248, 266, 321, 397, 442, 542
Callback Functions.....346
Calling Convention.....369
Calling Conventions.....124
Camel-casing.....29
Captured Variable.....450
Cardinal.....62, 547
Caret.....168
Cary Jensen.....6
Case.....35, 93p.
Case-insensitivity.....29
Catch.....266
Cdecl.....124
Char.....62, 67pp., 182
Characters.....67
Chars[].....190p., 195
Checkbox.....91
Checking Memory.....381
Chr.....69, 86
Chris Bensen.....295
Class.....36, 547
Class Completion.....213
Class Constraints.....413
Class Constructors.....41, 349, 411
Class Data.....343
Class Helpers.....357
Class Methods.....343
Class Of.....352
Class Operator.....155
Class Properties.....348
Class References.....352
Class Var.....344
Classes.....212
Classes Unit.....527
ClassInfo.....506
ClassName.....353, 505p.
ClassNameIs.....506
ClassParent.....506
ClassType.....505p.
Clear.....389
ClientDataSet.....429
Closing.....535
Closures.....301, 446, 546
COBOL.....540
Code.....19

Code Completion.....	33, 289p., 294, 304
Code Insights.....	33
Code Parameters.....	111, 117
Code Point.....	547
Code Points.....	175
CodeGear.....	542
CodeRage.....	6
COM.....	67, 115, 164, 207, 295, 322p., 337, 547
Comma.....	94
Comments.....	25
Common Ancestor Class.....	243
Comp.....	71
Compare.....	194
Comparison Operators.....	81
Compiler Directive.....	
Example.....	274
RTTI.....	468
\$ALIGN.....	147p.
\$DEFINE.....	46, 384
\$ELSE.....	46
\$HIGHCHARUNICODE.....	185
\$IF.....	47p.
\$IFDEF.....	47, 49, 127
\$IFDEF and \$IFNDEF.....	46
\$IFEND.....	47p.
\$IFNDEF.....	46
\$INCLUDE.....	43, 49
\$INLINE.....	121
\$J.....	59
\$M.....	298p., 465, 520
\$RTTI.....	468p.
\$SCOPEDENUMS.....	77
\$StrongLinkTypes.....	470
\$VARPROPSETTER.....	295
\$WeakLinkRTTI.....	469
\$Z.....	76
\$ZEROBASEDSTRING.....	192
Compiler Directives.....	26, 46, 547
Compiler Versions.....	47
ComponentCount.....	521
Components.....	521, 547
Compound Statements.....	90
Concatenating Strings.....	192
Concatenation.....	82
Conditional Defines.....	46
Console Application.....	20
Const.....	35, 372
Const Reference.....	116
Constant Parameters.....	115
Constants.....	58
Constructor.....	36
Constructors.....	228, 252, 278, 316, 388
Contains.....	194
Continue.....	102p.
Control Characters.....	68
Controls.....	548
ControlsEnum.....	
Example.....	363
ControlsEnum Application Project.....	363
Conversion.....	55
Conversions.....	85
ConvertFromUtf32.....	183
Copy.....	138, 194
Copy-on-write.....	187p.
CountChars.....	194
Covariant Return Types.....	442
CPU.....	70, 87, 101, 548p.
Create.....	243, 388, 504, 523
Creating A Component.....	309
Currency.....	71
Current.....	312
D	
Data Types.....	61, 548
Date.....	83p.
DateTimeToStr.....	84
David I.....	6
DayOfWeek.....	84
De Morgan's Law.....	100
Debugger.....	267
Dec.....	65, 182
DecodeDate.....	84
Default.....	294, 408
Default Constructor Constraint.....	418
Default Parameters.....	119
DefaultTextLineBreakStyle.....	514
DefaultUnicodeCodePage.....	202
Delayed Loading.....	128
Delegation.....	301, 303
Delete.....	139
Delphi.....	4, 541, 547, 551
DeQuoted.....	195

560 - C: Index

Design Patterns.....548
Design-time.....297
Destroy.....229, 243, 374
Destructor.....36, 229, 374
Destructors.....388
Dictionary.....427
Dispose.....169
DisposeOf.....376
Div.....81p.
DLL.....549
Do.....35, 266
DoCompare.....425
Dotted Unit Names.....40
Double.....35, 70
Downto.....35, 95
DupeString.....197
Dynamic.....253, 307
Dynamic Arrays.....136, 139
Dynamic Binding.....248
Dynamic Link Library.....549

E

Early Binding.....248
Editor Colors.....32
EDivByZero.....268
EExternalException.....129
EInvalidCast.....259
Elixir.....110
Else.....35, 91
Embarcadero Technologies.....4p., 542, 551
Empty.....187
Encapsulation.....150, 220, 243, 286, 299
EncodeDate.....84
End.....34
EndOfStream.....528
EndsWith.....194
Enumerated Types.....76
Enumeration.....311
EProgrammerNotFound.....269
Equals.....194, 508
Erich Gamma.....548
Erik Van Bilsen.....275, 441
Erlang.....110
Error Insight.....33
EurekaLog.....272
Event-Driven Programming.....300

Events.....300pp., 304p., 315, 549
Example.....
 AdvancedExcept.....280p., 284
 AlignTest.....147
 Animals1.....247pp.
 Animals2.....249p.
 Animals3.....255
 AnonAjax.....460pp.
 AnonButton.....454
 AnonLargeStrings.....456
 AnonymFirst.....446, 449, 451
 ArraysTest.....133p.
 AutoRTTI.....299
 BinaryFiles.....528
 CaseTest.....93
 CharsTest.....68p., 96
 CharTest.....183p.
 ClassConstraint.....413
 ClassCtor.....350
 ClassHelperDemo.....358
 ClassRef.....354, 356
 ClassStatic.....346, 348
 ClicksCount.....224
 CodePoints.....176
 ControlHelper.....359
 CountObj.....348
 CreateComps.....226, 228
 CustomerDictionary.....427
 Date3.....229p.
 DateComp.....309
 DateComponent.....311
 DateCompTest.....311
 DateEvent.....307, 311
 DatePackage.....310p.
 DateProperties.....292
 Dates1.....215
 Dates2.....222
 Dates3.....231
 Dates4.....233
 DerivedDates.....241
 DynamicEvents.....304
 DynArray.....137p.
 DynArrayConcat.....139
 EncodingsTest.....204
 ErrorLog.....277
 ExceptFinally.....273

ExceptionFlow.....	271	ReaderWriter.....	527
ExceptionsTest.....	266	RecordMethods.....	151, 153
ExpressionsTest.....	80	RecordsDemo.....	143
FloatTest.....	71p.	RecordsTest.....	145, 147
FlowTest.....	102, 113	ReintroduceTest.....	252
FormatString.....	199	RttiAccess.....	480
FormProperties.....	290	RttiAttrib.....	483p.
ForTest.....	96, 98	RttiIntro.....	467
FunctionsTest.....	107	SafeCode.....	388, 390, 393
FunctionTest.....	106, 109p.	ShowMemory.....	382
GenericClassCtor.....	411	ShowUnicode.....	179
GenericCodeGen.....	406	SmartPointers.....	438
GenericInterface.....	432p.	SmartPointersMR.....	438
GenericMethod.....	405	StaticCallBack.....	347
GenericTypeFunc.....	409p.	StringHelperTest.....	195
HelloConsole.....	20	StringListVsDictionary.....	430
HelloPlatform.....	49	StringMetaTest.....	201p.
HelloVisual.....	23	Strings101.....	187
IdentifiersTest.....	28	TimeNow.....	84
IfTest.....	91p.	TypeAliasHelper.....	366
InliningTest.....	122	TypeCompRules.....	401, 403
IntegersTest.....	64, 66	TypesList.....	470, 474
InterceptBaseClass.....	488	VariablesTest.....	55, 58
Intf101.....	323, 325	VariantTest.....	165p.
IntfConstraint.....	415, 417p., 433	VarProp.....	295
IntfContraints.....	432	ViewDate.....	234, 241, 292
IntfDemo.....	331p., 336	VisualInheritTest.....	261
IntfError.....	327p.	WebFind.....	458p., 461
IoFilesInFolder.....	524	XmlPersist.....	495
KeyValueClassic.....	398	Except.....	37, 266p., 269, 273
KeyValueGeneric.....	400	Exception.....	281p., 351
LargeString.....	193, 456	Exception Handling.....	265
LeakTest.....	383	Exceptions.....	388
ListDemoMd2005.....	421, 423	Exceptions Handling.....	37
LoopsTest.....	99	Exceptions Hierarchy.....	268
NestedClass.....	237	Exclude.....	79
NestedTypes.....	236	Exit.....	103, 112p.
NumbersEnumerator.....	312	Explicit.....	155p.
ObjFromIntf.....	338	Expression Context.....	446
OpenArray.....	140p., 143	Expressions.....	79p.
OperatorsOver.....	156, 158	Extended.....	70
OverloadTest.....	117pp.	Extended RTTL.....	468
ParamsTest.....	113pp.	External Functions.....	127
PointersTest.....	168, 170		
ProcType.....	126	F	
Protection.....	244p.	Fabrizio Schiavi.....	3

562 - C: Index

False.....67
Fastcall.....124
FastMM4.....382, 384
Fields Alignments.....147
File.....35
File Access.....523
File System.....549
File Types.....171
Files.....
 DCU.....43
 DFM.....297, 319
 DPR.....23, 45
 FMX.....297, 319
 INC.....43
 INI.....532
 PAS.....23, 43
Final Methods.....257
Finalization.....34, 41, 350
Finally.....37, 266, 272p., 373, 388p.
FindComponent.....522
FindHInstance.....392
FindType.....470
FireDAC.....49
FireMonkey.....21, 543, 547, 549
Floating Point.....70
FloatToDecimal.....86
FloatToStr.....86
FMXLinux.....549
For.....35, 94pp., 311
For-in.....97
Form.....315p., 549, 556
Format.....87, 141, 194, 198
FormatDateTime.....84, 167
FormatFloat.....87
Forms.....290
FORTRAN.....540
Forward.....108
Forward Declarations.....108
FPU.....70, 548p.
Free.....217, 230, 243, 279, 374
FreeAndNil.....219, 376
FreeInstance.....387
FreeMem.....169, 385
Friend Classes.....221
From.....479
Function.....36, 106, 550

Function Pointer.....125

G

Gang Of Four.....548
Garbage Collection.....217, 367
Generic Constraints.....412
Generic Containers.....421
Generic Dictionary.....426
Generic Methods.....404
Generic Type Declaration.....403
GetDirectories.....524
GetEnumerator.....98, 312p.
GetFiles.....524
GetHashCode.....194, 491, 509
GetMem.....169, 370, 385
GetMemoryManagerState.....381
GetMemoryMap.....381
GetMinimumBlockAlignment.....383
GetNumericValue.....183
GetPackages.....476
GetPropValue.....300
GetType.....470
GetTypeKind.....408
GetUnicodeCategory.....183
GetUserName.....128
GetWindowText.....385
Global Memory.....368, 550
Global Variables.....55, 60, 316, 368
Google.....545
Goto.....103
Graphemes.....175
GUI.....546, 548, 550
GUID.....323, 336, 432

H

Halt.....103
Haskell.....110
HasWeakRef.....408
Heap.....186, 370, 550
HFAAttribute.....516
High.....65, 133p., 137, 182, 192
HPPGENAttribute.....516

I

IComparer.....423, 425, 434
 IDE. 20, 25, 117, 151, 260, 289, 298, 309p., 541, 551
 Identifiers.....28p.
 IDispatch.....515
 IEqualityComparer.....434
 If.....35, 91p.
 IfThen.....180, 197
 IInterface.....322pp., 333, 336, 515
 IInvokable.....515
 Implementation.....34, 39
 Implements.....333, 515
 Implicit.....155p., 159, 440, 478
 In.....35, 78
 Inc.....65, 182, 289
 Include.....43, 79
 Indentation.....31
 Indexers.....294
 IndexOf.....194
 IndexOfAny.....194
 Information Hiding.....220
 Inheritance.....239, 246, 252, 551
 Inherited.....36, 231, 253, 263
 InheritsFrom.....353, 391, 506p.
 Initialization.....34, 41, 350
 Inline.....36, 121
 Inline Variables.....56, 95
 InnerException.....280p., 283
 Insert.....139, 194, 530
 InstanceSize.....353, 413, 505p.
 Int.....86
 Int64.....62
 Integer.....35, 62
 Integer Type Helper.....63
 InterestPayment.....72
 Interface.....34, 36, 39
 Interface Constraints.....415
 Interface Delegation.....332
 Interface ID.....432
 Interface Properties.....331
 Interfaces.....321p., 326, 339, 551
 Internet Of Things.....533
 Interposer Class.....340, 357
 Intrinsic Record Helpers.....64

Intrinsic Type Helpers.....4
 IntToStr.....87
 Invalid Typecast.....478
 Invoke.....480
 IO.....209
 IOS.....48, 543, 552
 IOS.....549
 Is.....258p., 336p., 390
 IsChecked.....91
 IsControl.....183
 IsDelimiter.....194
 IsEmpty.....194
 IsInArray.....183
 IsLetter.....68
 IsLetterOrDigit.....183
 IsLower.....183
 IsManagedType.....408
 IsNullOrWhiteSpace.....194
 IsNumber.....68, 183
 ISO Encodings.....174
 IsPointerToObject.....393
 IsPunctuation.....68
 IsSurrogate.....183
 IsUpper.....183
 IsWhiteSpace.....183
 IUnknown.....323

J

Java. 3, 25, 29, 51, 54, 73, 81, 94, 106, 132, 153, 214, 216, 226, 237, 243, 248, 257, 266, 286p., 297, 302, 321, 341, 542
 JavaScript 4, 25, 29, 51p., 54, 99, 106, 132, 148, 164, 209, 226, 301, 446, 542
 JclDebug.....272
 John Thomas.....6
 Join.....194
 JSON.....532

K

Key-value Pair.....398
 Keywords.....33

L

Label.....234

564 - C: Index

Language Keywords.....	33
LargeInt.....	63
LastIndexOf.....	194
Late Binding.....	248
Lazy Initialization.....	374, 428
Leak Detection.....	381
Length.....	137, 187
Library.....	34
Library Path.....	43
Lifetime.....	60
Lifetime Of Local Variables.....	448
LIFO.....	369
Linux.....	49, 549
ListBox.....	422
ListView.....	179, 429
Literal Values.....	53
Literals.....	184
Little Endian.....	204
LLVM.....	543
Loaded.....	523
LoadFromFile.....	531
Local Variables.....	60
Loops.....	97, 99p.
Low.....	65, 133p., 137, 182, 192
LowerCase.....	194
Lua.....	209
Lvalue.....	54

M

MacOS.....	48, 70, 549
MadExcept.....	272
Malcolm Groves.....	484
Malloc.....	387
Marco Cantu.....	6
Mastering Delphi.....	6
Max.....	122
MeanAndStdDev.....	72
MediaPlayer.....	250
Memo.....	22
Memory Leak.....	217
Message Handlers.....	254
MessageDlg.....	109
Metaclasses.....	353
Method.....	111
Method Chaining.....	530
Method Pointers.....	301

Methods.....	150, 215, 552
Methods Aliases.....	334
Microsoft.....	541, 556
MobilePreview.....	23
Mod.....	81
Modula-2.....	541
MomentSkewKurtosis.....	72
Monitor Synchronization.....	514
More On Weak References.....	377
Move.....	515
Multi-Dimensional Static Arrays.....	134
Multiple Inheritance.....	321
Multiplicative Operators.....	81

N

Name.....	315, 522
Named Constructors.....	229
Named Types.....	73
Namespaces.....	44
NativeInt.....	63
NativeUInt.....	169
Nested Exceptions.....	280
Nested Types.....	236
NeverSleepOnMMThreadContention.....	383
New.....	169
NewInstance.....	387
Nick Hodges.....	320
Nicklaus Wirth.....	89, 541
Niklaus Wirth.....	540
Nil.....	37, 166, 218, 375
Not.....	81
Notification.....	426
Now.....	84
NULL.....	166
Null Statement.....	90
NumberBox.....	93
Numeric Types.....	62

O

Object.....	36, 552
Object Inspector.....	304, 306, 308, 310
Object Reference Model.....	216, 371
Object-oriented Programming.....	209, 211
Objective-C.....	25, 81, 153, 164
Objects.....	212, 215

Objects As Parameters.....	371
Odd.....	65
Of.....	35
Of Object.....	302, 306
Ointers.....	148
On.....	266
OnChange.....	355
OnClick.....	303p., 306, 315, 455
OnCreate.....	179, 347
OnException.....	270, 277
OnMouseDown.....	355
OOP.....	551p.
Open Array Parameters.....	139
Open-closed Principle.....	239, 357
Operator Overloading.....	155
Operators.....	37, 79p.
Or.....	81
Ord.....	65, 86, 182
Ordinal Type.....	552
Ordinal Types.....	62
Out.....	115
Out-Of-Range.....	66
Overflow Checking.....	67
Overload.....	36, 116, 309
Overloaded Methods.....	231
Overloading.....	550
Override.....	36, 231, 249p., 309
Owner.....	373, 521
Ownership.....	521

P

Package.....	34
PadLeft.....	194
Parallel Programming Library.....	459, 533
ParamCount.....	515
Parameters.....	111
Parametric Type.....	399
ParamStr.....	515
Parent.....	227, 355
ParentClass.....	353
Parse.....	65, 194
Pascal.....	124, 539p.
Peter Wood.....	3, 5
PHP.....	164, 542
Piacenza.....	2
Pierre La Riche.....	382

Pointers.....	168, 553
Polymorphism.....	246, 248, 336, 553
Polytechnic Of Zurich.....	540
Position.....	525
Precedence.....	80
Pred.....	65
Pretty-printing.....	24
Private.....	36, 152, 219, 221, 244
Procedural Types.....	125
Procedure.....	36, 106, 553
Program.....	34
Program File.....	45
Project Manager.....	45, 310
Project Options.....	45p., 76, 553
Properties.....	286, 305, 319, 348, 553
Properties By Reference.....	295
Property.....	37
Protected.....	36, 219p., 243p.
Protected Hack.....	244
Public.....	36, 219p., 298
Published.....	36, 296pp., 465, 520
Python.....	164

Q

QualifiedClassName.....	506
QueryInterface.....	336
QuotedString.....	195
Quotes.....	53p.

R

RAD.....	553
RAD And OOP.....	314
RAD Studio.....	4
Raise.....	37, 266, 271
RaiseOuterException.....	281
Random.....	515
Randomize.....	515
Range Check Error.....	76
Range Checking.....	133
RawByteString.....	208
Read.....	37, 286, 525, 527p.
Read-only Property.....	287
Readers.....	527
Real.....	35, 70
Rebol.....	80, 209

566 - C: Index

Record.....35, 553
Record Helper.....364
Record Helpers.....363
Record Type.....143
Records Vs. Classes.....219
Records With Methods.....150
Recursion.....109, 554
RefAttribute.....516
Reference.....554
Reference Counting.....326
Reference Parameters.....113
Reference Types.....453
Reference-counting.....187
Reflection.....297, 465, 554
Register.....29, 124, 309
RegisterClasses.....320
RegisterComponents.....310
RegisterExpectedMemoryLeak.....383
Reintroduce.....252, 309, 316
Relational Operators.....81
Remove.....194, 530
Repeat.....35, 99
Replace.....194, 196, 530
ReportMemoryLeaksOnShutdown.....383
Repository On GitHub.....5
ResemblesText.....197
RestoreCursor Example.....274
Result.....106, 112
Return Type.....106
Return Values.....111
Reverse For.....95
ReverseString.....197
Robust Applications.....387
Round.....86p.
RTTI.....296, 298pp., 319, 466, 554
RTTI Classes.....472
Ruby.....80, 164
Rudy Velthuis.....72
Run Time Type Information.....466
Run-Time Library.....554
Rvalue.....54

S

SaveToFile.....531
Scope.....43
Scoped Enumerators.....76

Screen.....317
SDK.....555
Sealed.....36
Sealed Classes.....257
Search Options.....77
Search Path.....43, 555
Self.....152p., 225p., 301, 317, 345, 375
Semicolon.....90
Sender.....305, 455
Set.....35, 77
Set Operators.....78
SetLength.....136
SetMinimumBlockAlignment.....383
SetTimer.....347
Shared Object.....549
Shl.....81
ShortCut Key.....
 Ctrl+/......25
 Ctrl+C.....213
 Ctrl+D.....25
 Ctrl+Shift+C.....151, 290p.
 Ctrl+Shift+C.....289
 Ctrl+Shift+G.....323
 Ctrl+Shift+L.....60
 Ctrl+Shift+Up.....151
ShortInt.....62
ShortString.....200, 207
ShowMemory.....382
ShowMessage.....23
Shr.....81
Single.....35, 70
Singleton Pattern.....327, 351
Singly-rooted Class Hierarchy.....504
Size.....64, 525
SizeOf.....65, 144, 147, 408, 413
Slice.....141
SLineBreak.....514, 530
SmallInt.....62
Smalltalk.....164, 212
Smart Pointers.....435
Sort.....423
Source Code.....5
Split.....195
Square Brackets.....190, 481
Stack.....271, 369, 555
Stack Overflow.....109

Standard Template Library.....	421
StartsWith.....	194
Statements.....	89
Static Arrays.....	132
Static Class Methods.....	346
Stdcall.....	124
Steve Tendon.....	397
StoredAttribute.....	516
Streaming.....	522
Streams.....	525
Strict.....	37
Strict Private.....	220
Strict Protected.....	220
String.....	35, 186
String Concatenation.....	187
String Helper.....	194
String Lists.....	529
StringRefCount.....	201
StrToDateTime.....	84
StrToFloat.....	86
Structure Of A Program.....	38
Subclassing.....	240
Subrange.....	75
SubString.....	194
Succ.....	65
Support.....	336
Swift.....	251
Switch.....	94
Synchronize.....	458
Syntax.....	24
Syntax Highlighting.....	32

T

TabControl.....	179
Table Of Contents.....	7
Tag.....	390, 522
TAggregatedObject.....	332, 515
TBasicAction.....	519
TBinaryReader.....	528
TBinaryWriter.....	528
TBits.....	518
TBufferedFileStream.....	526
TButton.....	355, 384, 422
TBytesStream.....	526
TCharacter.....	182
TCharHelper.....	182
TClass.....	354
TCollection.....	518
TComparer.....	423, 425
TComponent.....	309, 316, 327, 519p.
TContainedObject.....	515
TCustomAttribute.....	482, 516
TDataModule.....	519
TDate.....	83
TDateTime.....	83, 85, 87, 222, 514
TDictionary.....	421, 430
TDirectory.....	523
TEdit.....	320, 355
Template Classes.....	397
TEncoding.....	203p., 208
TextFile.....	171
TFile.....	523
TFileStream.....	526
TFilterPredicate.....	524
TForm.....	260
TFormatSettings.....	85
TFunc.....	453
TGUID.....	360, 514
THandle.....	514
The Delphi Magazine.....	315, 340
THeapStatus.....	514
Then.....	35
This.....	153
Threads Synchronization.....	457
Throw.....	266
Time.....	83p.
Timer.....	85
TInterfacedObject.....	324pp., 333, 515
TInterfaceList.....	518
TList.....	388, 390, 421, 518
TMemoryManagerEx.....	514
TMemoryStream.....	526
TMethod.....	302, 514
TMonitor.....	514
To.....	35
TObject. 217, 229, 243, 253, 259, 278, 322, 353,	374, 503p.
TObjectDictionary.....	421, 426
TObjectList.....	391, 421p., 426
TObjectQueue.....	421, 426
TObjectStack.....	421, 426
ToCharArray.....	194

568 - C: Index

ToInteger.....194
ToLower.....68, 183, 194
Tools Palette.....310
ToString.....64, 67, 87, 282, 508
ToUpper.....68p., 183, 194
TPath.....523, 525
TPersistent.....298p., 372, 518, 520
TProc.....453
TQueue.....421
TResourceStream.....526
Trial Version.....5
Trim.....194
TRttiContext.....470, 473, 476
TRttiObject.....472
TRttiType.....471
True.....67
Trunc.....86
Try.....37, 266, 272p.
TryParse.....65
TSingletonImplementation.....326p., 434
TStack.....421
TStopWatch.....122
TStream.....519, 525
TStreamReader.....527p.
TStreamWriter.....527
TStringBuilder.....192, 529p.
TStringList.....372, 430, 519, 531
TStringReader.....527
TStrings.....294, 519, 531
TStringStream.....526
TStringWriter.....527
TTextLineBreakStyle.....514
TTextWriter.....495
TThread.....458, 519
TTime.....83
TUnicodeBreak.....182
TUnicodeCategory.....182
Turbo Pascal.....540p., 551
TValue.....477, 479
TVarData.....142, 166
TVarRec.....142
TVirtualMethodInterceptor.....487
TVisibilityClasses.....468, 514
Type.....35
Type Aliases.....365
Type Cast Operators.....258

Type Compatibility.....246
Type Compatibility Rules.....403
Type Derivation.....240
Type Inference.....57
Type Name Equivalence.....74
Type Promotions.....159
Type-Variant Open Array Parameters.....141
Typecasting.....85
TypeInfo.....408
TypeScript.....211

U

UCS4Char.....70, 177, 182
UCS4String.....207
UInt64.....62
UIntPtr.....168p.
Unary Operators.....81
Unicode.....29, 53, 68p., 174, 179, 203, 555
Unicode Transformation Formats.....177
Unit.....34, 38
 Generics.Collections.....421, 426
 Generics.Defaults.....326, 423, 434
 System.....63, 83, 166, 171, 503, 513
 System.Actions.....531
 System.AnsiStrings.....531
 System.Character.....68, 70, 182, 531
 System.Classes.....518, 527, 531
 System.Contnrs.....421, 532
 System.ConvUtils.....532
 System.DateUtils.....83, 532
 System.Devices.....532
 System.Diagnostics.....122, 532
 System.Hash.....532
 System.ImageList.....532
 System.IniFiles.....532
 System.IOUtils.....77, 523, 525, 532
 System.JSON.....532
 System.Math.....72, 116, 122, 532
 System.Messaging.....532
 System.NetEncoding.....532
 System.RegularExpressions.....532
 System.Rtti.....470, 532
 System.StrUtils.....181, 197, 532
 System.SyncObjs.....532
 System.SysUtils..83, 85, 181, 197, 203, 268, 351, 364, 453, 533

System.Threading.....	533
System.Types.....	533
System.TypeInfo.....	299, 409, 470, 533
System.Variants.....	533
System.Zip.....	533
Winapi.Windows.....	127
Unit Name.....	39
Unit Scope Names.....	40
UnitName.....	506
Unnamed Types.....	73
Unsafe.....	380
Unsafe References.....	328
UnsafeAttribute.....	516
Until.....	35
UpCase.....	183
UpperCase.....	29, 194
User-Defined Data Types.....	72
Uses.....	34, 42, 45
UTF-16.....	177, 203
UTF-32.....	177
UTF-8.....	177p., 203, 205
UTF32Char.....	207
UTF8String.....	206p.
V	
Var.....	35, 52, 114
Variables.....	52
Variant.....	142
Variant Records.....	146
Variants.....	164pp.
VCL.....	547, 556
Virtual.....	36, 249p., 253
Virtual Class Methods.....	345
Virtual Method Table.....	253
Virtual Methods.....	556
Virtual Methods Interceptors.....	486
Visibility.....	43, 60
Visual Basic.....	148
Visual Component Library.....	21
Visual Form Inheritance.....	260
VmtInstanceSize.....	393
VmtSelfPtr.....	392
VolatileAttribute.....	516
VType.....	166

W

Weak References.....	328, 379
WeakAttribute.....	516
While.....	35, 99p.
White Space.....	30
WideChar.....	67, 70
WideString.....	207
Windows.....	48, 545, 549, 556
Windows API.....	127, 254, 346
With.....	36, 148p.
WM_USER.....	254
Word.....	62
Write.....	37, 286, 525, 527p.
Writers.....	527

X

XML Doc.....	26
XML Streaming.....	495
Xor.....	81

—

_AddRef.....	325p.
_Release.....	325p.

:

:=.....	54
---------	----

@

@.....	81
--------	----

#

#.....	184
--------	-----

=

=.....	54
==.....	54

\$

\$IF.....	46
-----------	----

€	€.....175, 184
----------	----------------